N° d'ordre: 3707

THÈSE

PRÉSENTÉE À

L'UNIVERSITÉ BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET INFORMATIQUE

Par Laurent Sagaspe

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : Informatique

Allocation sûre dans les systèmes aéronautiques : Modélisation, Vérification et Génération

---

Soutenue le :	4 Décembre 2008	
Après avis de :	Fabrice Kordon	**Rapporteurs**
	Jean-Paul Bodeveix	**Rapporteurs**

Devant la Commission d'examen formée de :

Jean-Michel Couvreur	Professeur des universités (LIFO)	**Directeur de Thèse**
Pierre Bieber	Ingénieur de recherche (ONERA)	**Co-Directeur de Thèse**
Alain Griffault	Maître de Conférences (LaBRI)	**Examinateurs**
Jean-Philippe Domenger	Professeur des universités (LaBRI)	**Examinateurs**
Igor Walukiewicz	Directeur de Recherche CNRS (LaBRI)	**Examinateurs**
Fabrice Kordon	Professeur des universités (LIP6)	**Rapporteur**
Jean-Paul Bodeveix	Professeur des universités (IRIT)	**Rapporteur**

– 2008 –

Remerciements

Voilà, c'est avec beaucoup de plaisir et d'émotion que je rédige ces derniers mots qui signent la fin de mon aventure de doctorant...

Je tiens à remercier en premier lieu Fabrice Kordon et Jean-Paul Bodeveix pour avoir accepté d'être les rapporteurs de mes travaux de thèse, ainsi que Igor Walukiewicz, Jean-Philippe Domenger et Jean-Michel Couvreur pour avoir accepté de constituer mon jury de thèse. Chacun d'entre eux m'a apporté de précieuses critiques enrichissantes pour mes travaux.

Je tiens également à remercier chaleureusement Alain Griffault pour ses conseils et l'attention toute particulière qu'il a su me donner. Sa gentillesse et sa disponibilité ont permis de minimiser la distance Bordeaux - Toulouse.

Mes plus sincères remerciements s'adressent à Pierre Bieber, sans qui cette thèse n'aurait vu le jour, car il a initié ces travaux et m'a donné la chance de travailler à ces cotés (et oui 3 ans à partager le même bureau). Ses compétences m'ont toujours impressionné, mais grâce à son attention, sa pédagogie, sa bonne humeur et surtout son optimisme, il a réussi à transformer cette difficile épreuve de thèse en années inoubliables d'excellents souvenirs à ses côtés. Il a su trouver les bons mots pour faire avancer les choses et dans les moments difficiles, sa bonne humeur et son optimisme ont été décisifs pour ma motivation. Ses enseignements resteront gravés en moi, qui resterai enrichi de ses explications, de ses nombreux coups de crayon et dessins indélébiles.

Encore mille fois merci pour tout Pierre !

Comment parler de l'ambiance de ce bureau sans penser à tous les acteurs du département qui y ont participé... Je pense à Christel qui a toujours été de très bon conseil, sa gentillesse, sa bonne humeur et sa disponibilité m'ont rendu beaucoup de service. Son perfectionnisme (tant redouté par les thésards lors des répétitions) est tellement constructif et enrichissant qu'il m'a beaucoup profité lors de mes présentations. Je pense également à Virginie qui par ses histoires savait nous donner un bon fou rire pour la journée. Ainsi que tous ceux qui prenaient le temps de venir partager un moment bavard dans le bureau : Bruno, Charles, Chritiane, Claire, Frédéric, Guy, Josette, Jean-Loup, et Jean-Yves. Et merci à Jacques Cazin pour son accueil dans le département.

Puis un laboratoire c'est aussi une ambiance crée par ces jeunes locataires, les thésards : Je pense à Christophe pour notre amitié pendant et après la thèse, depuis Balma jusqu'a Menecy, un grand merci pour avoir tenté de venir assisté à ma soutenance et m'avoir mis dans un état de panique 15 minutes avant le début de ma soutenance à cause d'une Velsatis. Je pense également à Alex qui fut un adversaire redoutable à *Tower defense*, à Matthieu qui a su construire la map du CERT, à Thomas qui restera toujours le stagiaire casqué, à Julien, Sophie H, Sophie L et Stéphanie et les nouveaux arrivants Cédric et Romain toujours partant pour faire une pause.

Les physiciens « d'en face » qui ont également participé à la bonne humeur de toutes les pauses, je pense à Domingo avec qui on a réussi (pas longtemps) à faire découvrir le Mus et ainsi détrôner les parties endiablées de coinche. Je pense à Maud et Éric fervent défenseur des Bretons. Je pense enfin à Flo, Rémy et Hélène pour les tournois de coinche. Il reste Manu mon partenaire de Squash qui avec son short et sa gentillesse a su me donner envie de faire du théâtre.

Quelques mots aussi pour remercier mes collègues d'APSYS qui dans les derniers mois de cette

thèse m'ont encouragé et permis de terminer dans de bonnes conditions. Merci à Nico, Pierre S., Pierre M., Amandine, Julien, Xavier, Alexis, Sébastien, François et Valérie. Je tiens également à remercier Jean pour sa confiance et la chance qu'il m'a donné en me permettant de travailler sur des sujets proches de ma thèse.

Je tiens également à remercier ma famille. Mon frère pour m'avoir accompagné et soutenu pour ma dernière épreuve de vie de Thésard : La soutenance ! Mes parents pour leur soutien : je pense aux relectures courageuses de mon père malgré son incompréhension du domaine et des moyens techniques défavorables au fin fond du Pays Basque et je pense à la fierté de ma mère de me voir aller jusqu'au bout. Enfin, je pense très fort à mes grands parents Papou, Mamette, Aïtaxi, Amatxi qui de près ou de loin m'ont toujours accompagné.

Je n'oublie pas la famille Gaudan pour leur compréhension et leur soutien dans les deniers mois. C'est aussi grâce à eux que la soutenance a pu se dérouler dans les meilleures conditions possibles.

La dernière personne et non la moindre que je souhaite remercier de tout mon coeur est ma moitié. Elle a su toutes ces années trouver les mots pour m'encourager et me soutenir (surtout dans les moments difficiles), elle a toujours su m'épauler et s'occuper de tout pour que je puisse consacrer le reste de mon temps à bien terminer. Je profite de ces quelques lignes pour la remercier de tout l'amour qu'elle a pu me donner.

À tous ceux que j'oublie un grand merci.

Neke ondoren, poza

à Guillaume et Papou.



Table des matières

Table des figures

Première partie

Introduction au domaine

Chapitre 1

Introduction

1.1 Contexte général - Sécurité et fiabilité des systèmes aéronautiques

Le contexte applicatif dans lequel s'inscrivent nos travaux est celui de la conception et la validation de systèmes avioniques ayant un rôle critique dans le fonctionnement d'un avion (système hydraulique, système électrique, système de gestion de vol, etc.). Ces systèmes sont comme considérés critiques du point de vue de la sécurité (aussi nommée sécurité-innocuité dans [Lap96]) car leur dysfonctionnement est susceptible d'avoir des conséquences catastrophiques sur les personnes (l'équipage, les passagers ou les personnes au sol), les biens (l'avion, l'aéroport, ...) ou l'environnement. Une fois embarqués dans les avions, ces systèmes doivent également présenter une forte fiabilité qui, selon [Lap96], est définie par l'aptitude d'un système à assurer sa fonction pendant sa durée de vie, c'est-à-dire tout au long de la durée d'exploitation d'un avion, qui avoisine les 50 ans. Bien sûr, certains composants de ces systèmes seront amenés à être remplacés en cas de panne du fait d'usure matérielle, mais de manière générale, la plupart d'entre eux seront maintenus tout au long de la vie d'un avion.

Il est donc primordial que chacun des systèmes critiques composant un avion soit conçu en considérant les contraintes de fiabilité. Pour préciser les notions de sécurité et de fiabilité d'un système, des exigences lui sont associées et caractérisent son taux de défaillance acceptable (en probabilité). Par exemple, une fonctionnalité indispensable pour le bon déroulement d'un vol se verra associée une exigence dont le taux de défaillance acceptable sera suffisamment faible pour que la défaillance soit considérée non atteignable au cours des heures de vol effectuées dans la vie d'un avion.

Les taux de panne des composants constituant un système ne permettent généralement pas de garantir le taux global de défaillance du système. Par conséquent, le mécanisme de redondance des composants du système est souvent utilisé pour diminuer son taux global de défaillance. Ce mécanisme consiste à recourir à plusieurs composants matériels identiques pour supporter une même fonction.

Finalement, pour répondre à des exigences fonctionnelles caractérisées par une probabilité de défaillance, une allocation des fonctions sur une architecture physique doit être définie. Cette allocation doit garantir, en tenant compte (i) des taux de pannes associés à chacun des composants matériels et (ii) des impacts relatifs des pannes entre composants du fait de leurs relations, que le taux de défaillance global du système est acceptable.

Dans le but de garantir la sécurité et la fiabilité des systèmes critiques utilisés dans l'aéronautique, un processus de développement a été défini et doit être appliqué pour la conception de tout système

avionique embarqué. En effet, pour qu'un système soit certifié, et donc embarquable, il doit être prouvé que le processus en question, l'ARP[47696] a bien été appliqué. L'objectif est de maîtriser les différentes étapes du développement d'un système critique pour obtenir au final un système digne de confiance, c'est à dire satisfaisant les exigences qui lui sont associées.

1.2 Processus de développement des systèmes aéronautiques critiques

La plupart des industriels du domaine suivent les recommandations qui sont définies dans le document ARP4754[47696]. Bien entendu, chaque industriel est libre de définir un processus de développement spécialisé pour répondre aux spécificités de l'entreprise ou du type d'aéronef. Par exemple, la directive Airbus ABD200 [Air96] spécifie le processus dédié à la vérification de la sécurité des systèmes aéronautiques embarqués dans les avions Airbus.

Comme schématisé par la figure 1.2, ces recommandations s'organisent autour d'un processus d'évaluation de la sûreté de fonctionnement d'un système basé sur quatre analyses principales :
- FHA (Functional Hazard Assessment),
- PSSA (Preliminary System Safety Assessment),
- SSA (System Safety Assessment) et
- CCA (Common Cause Analysis).

Ces quatre analyses ne représentent en fait qu'une même analyse globale mais elles portent sur différents niveaux de maturité de la définition du système.

FHA Cette analyse est initiée au tout début du processus. Elle identifie les différentes situations redoutées (dites FC : *Failure Conditions*) associées aux différentes fonctions réalisées par les systèmes d'un avion. L'ensemble de ces situations à contrôler est construit à partir des défaillances considérées pour chaque constituant du système. Dans un premier temps, les seuls modes de défaillance à considérer sont la perte et le fonctionnement erroné des constituants du système. Ensuite chacune des situations redoutées est classée selon sa gravité. La définition de la classification de gravité (criticité) est donnée par le tableau 1.1. Ces niveaux de criticité se déclinent depuis le niveau *mineur* (impact peu conséquent sur le bon déroulement d'un vol) jusqu'au niveau *catastrophique*. Pour être acceptable, la probabilité d'une défaillance du type catastrophique ne doit dépasser 10^{-9} par heure de vol.

Classe	**Effet de la défaillance**	**Probabilité / Objectif**
Catastrophic (**CAT**)	L'avion ne peut voler de façon sûre, perte de l'avion, de l'équipage et des passagers	10^{-9}
Hazardous (**HAZ**)	Réduction importante des marges de sécurité ou des fonctions, augmentation trop importante de la charge de travail pour l'équipage, blessures fatales ou sévères d'un petit nombre de passagers	10^{-7}
Major (**MAJ**)	Réduction significative des marges de sécurité ou des fonctions, augmentation importante de la charge de travail pour l'équipage, inconfort pour les passagers, blessures possibles	10^{-5}
Minor (**MIN**)	Réduction légère des marges de sécurité ou des fonctions, augmentation de la charge de travail pour l'équipage, inconfort pour les passagers	10^{-3}

TAB. 1.1 – Classification des défaillances dans la **FHA**

La FHA décline les situations redoutées pour chaque phase de vol et détermine les différentes exigences à respecter afin de limiter les effets des défaillances qui sont considérées. A chaque situation redoutée est alors associé un objectif quantitatif relatif à sa classification (cf. TAB.1.1). Cette exigence

quantitative est complétée dans certains cas par un objectif qualitatif tel que : *Une panne simple ne doit pas conduire à une situation redoutée classée Catastrophique.*

L'objectif qualititatif est utile pour faciliter la vérification de la tenue des exigences quantitatives. En effet, la probabilité associée à l'occurrence d'une défaillance peut être inférieure à une fois par milliard d'heures de vol. Or il n'est pas possible de vérifier qu'un équipement respecte effectivement ce type d'exigences. En effet le cumul des heures de vol de tous les appareils Airbus est très inférieur au milliard d'heures de vol. Il faut donc fixer des exigences plus facilement vérifiables avec des taux de défaillance de l'ordre d'une fois toutes les 10 000 heures de vol (ce qui représente tout de même presque trois années d'utilisation pour un avion qui fonctionne en moyenne 10 heures par jour). C'est pour cette raison qu'il n'est pas souhaitable qu'une panne simple conduise à une situation catastrophique dont le taux de défaillance est de l'ordre de 10^{-9} par heure de vol.

Dans la suite de ce document, nous considérons une famille d'exigences qualitatives qui correspondent à la façon dont sont construits les systèmes aéronautiques : *Une combinaison de moins de N pannes ne doit pas conduire à une situation redoutée classée Sev,* avec $N = 3$ si $Sev = CAT$, $N = 2$ si $Sev = HAZ$ ou MAJ et $N = 1$ si $Sev = MIN$.

PSSA L'objectif de cette analyse est de démontrer que la liste des conditions de panne issue de la FHA est complète, et d'identifier de nouvelles exigences de sécurité. C'est lors de cette analyse que l'on identifie des stratégies de protection à mettre en place afin de tenir les objectifs. Une PSSA doit donc identifier les pannes et leurs combinaisons contribuant aux situations redoutées explicitées par la FHA. Les choix établis par les concepteurs de l'architecture du système sont ainsi vérifiés par des analyses de sécurité réalisées généralement à l'aide d'arbres de défaillance. L'arbre de défaillance est une représentation graphique de la formule logique exprimant la relation de causalité entre l'occurrence de défaillances des composants d'un système et la situation redoutée. Les analyses associées aux arbres de défaillances permettent de vérifier la tenue des exigences qualitatives et quantitatives. Les taux de défaillances des composants deviennent alors des exigences quantitatives à garantir par les fournisseurs d'équipements.

SSA Cette analyse consiste à démontrer que l'architecture matérielle finale du système est bien en accord avec les objectifs fixés. Il s'agit de vérifier la cohérence entre les taux de défaillance à atteindre et ceux présentés par les composants matériels du système. Contrairement à une PSSA, qui propose une méthode d'évaluation d'une architecture relativement à des exigences, la SSA *vérifie* qu'une architecture mise en place est conforme aux objectifs. Les taux de défaillance à prendre en compte pour la construction et l'analyse des arbres de défaillances sont ceux fournis par les différents fournisseurs. Les arbres sont mis à jour avec ces nouveaux taux de défaillance et sont ré-évalués afin de quantifier les situations redoutées définies au niveau avion et de s'assurer que les taux obtenus sont bien conformes à ceux identifiés dans la FHA.

CCA L'objectif de cette analyse est d'identifier les défaillances, dites de mode commun, qui sont susceptibles d'affecter plusieurs fonctions réalisées par un système.

Par exemple, deux fonctions dont les défaillances doivent être indépendantes ne doivent présenter aucune nouvelles défaillances de mode commun les impliquant toutes les deux. Par conséquent, une fonction redondée ne doit être impliquée dans aucune défaillance de mode commun impliquant ses fonctions de redondance pour accroître la sécurité et donc réduire le risque de défaillance.

L'utilisation d'un équipement commun (comme, par exemple, une source d'alimentation électrique) par plusieurs fonctions est une source de défaillances de mode commun. En effet, la défaillance de l'équipement commun va entrainer la défaillance des différentes fonctions qui l'utilisent.

Une autre source de défaillances de mode commun est liée à l'installation dans l'avion des équipements d'un système. L'impact sur les équipements de l'éclatement d'un pneu ou d'un moteur est calculé lors de l'analyse PRA (Particular Risks Analysis). Dans ce cas, on suppose que toutes les fonctions associées aux équipements impactés par un débris de pneu ou de moteur tombent en panne simultanément.

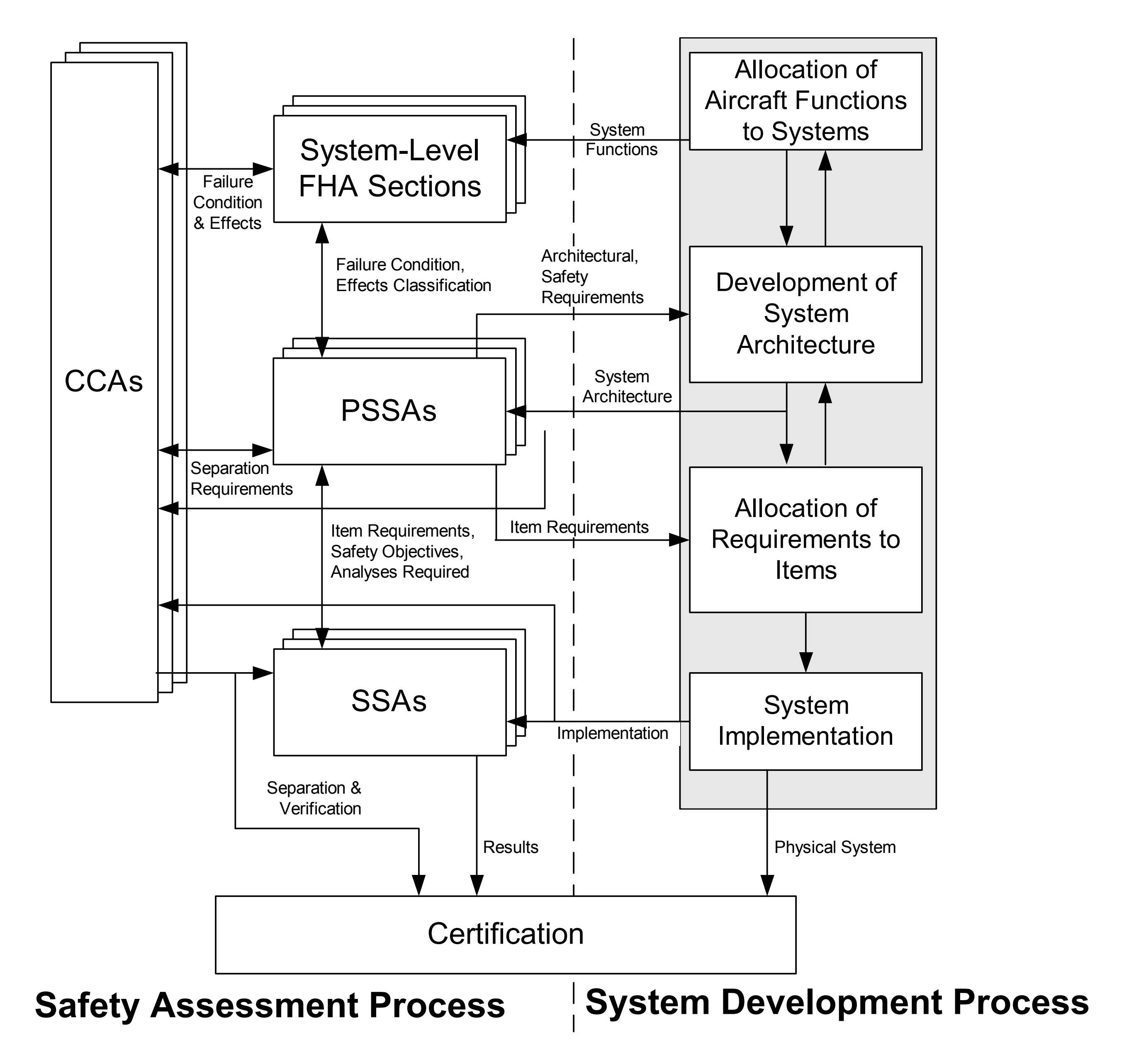


FIG. 1.1 – Cycle de développement selon l'ARP 4754

1.3 Limites du processus actuel

Le processus actuel repose en grande partie sur la définition et l'analyse d'arbres de défaillances. La définition des arbres de défaillances associés à un système est habituellement établie par des experts de manière manuelle. La première étape (de définition des arbres de défaillances) représente un travail très long. En effet, ce travail nécessite l'analyse de tous les documents de description du système de type fonctionnel pour la FHA, fonctionnel et architectural pour la PSSA et fonctionnel, architectural et composants matériels pour la SSA. Un premier ensemble de limites associées à cette approche est :

- la définition de l'arbre de défaillance est longue,
- il n'est pas aisé de tenir compte des modifications apportées dans la définition du système, c'est l'ensemble de l'arbre de défaillance qui doit être examiné pour déterminer ce qui doit être modifié,
- la représentation sous forme d'arbre est assez éloignée des diagrammes habituellement utilisés pour décrire les systèmes, ce qui ne facilite pas la compréhension (par les concepteurs des systèmes) des scénarios critiques trouvés par les analystes de sécurité.

Des projets de recherche récents [BVÅ+03] ont montré qu'il était possible de remédier aux limites que nous venons de mentionner en utilisant les méthodes formelles pour modéliser et analyser la sécurité des systèmes aéronautiques. Ces techniques permettent de :
- créer des bibliothèques de composants réutilisables, ceci contribue à diminuer l'effort et la durée consacrés à la production des modèles,
- générer les arbres de défaillances et évaluer la tenue des exigences qualitatives automatiquement à partir d'un modèle du système étudié, ceci permet d'évaluer et de comparer rapidement la sécurité de plusieurs solutions d'architectures,
- présenter les modèles sous une forme proche des diagrammes utilisés par les spécialistes des systèmes et
- simuler sur ces modèles les scénarios les plus critiques, ceci améliore le dialogue entre les analystes de sécurité et les concepteurs des systèmes.

Ces nouvelles techniques contribuent essentiellement à simplifier les analyses PSSA et SSA. En revanche elles assistent assez peu l'analyse CCA de recherche des modes communs de défaillance. Or, des évolutions récentes des architectures aéronautiques renforcent l'importance de l'analyse CCA. En premier lieu, l'avènement des architectures avioniques modulaires intégrées sur les avions civils récents (Boeing B77, Airbus A380 et A400M) conduit à partager des ressources de calcul et de communication entre plusieurs fonctions. Ce partage est une source de défaillances de mode commun. En effet, la défaillance d'un calculateur va entraîner la défaillance des différentes fonctions qui l'utilisent. En second lieu, le développement des avions tend à s'organiser autour de maquettes numériques d'avion qui rendent possible l'étude de l'installation des équipements dès les phases amont du développement des systèmes. Et, comme nous l'avons déjà vu dans la section précédente, l'installation des équipements au sein de l'avion est une autre source de défaillances de mode commun.

Plus généralement, durant plusieurs phases du développement des systèmes les ingénieurs doivent proposer des allocations : allocation des fonctions sur les équipements, allocation de ces équipements sur des emplacement de l'avion. Il faut que l'analyse CCA s'assure que ces allocations soient sûres c'est à dire qu'elles n'invalident pas les exigences de sécurité du système étudié.

1.4 Objectifs de la thèse et démarche

Nos discussions avec les ingénieurs en charge des analyses CCA ont abouti au constat suivant :
- un premier souhait qu'ils expriment serait de disposer de techniques permettant d'évaluer facilement l'impact d'une allocation sur la sécurité d'un système,
- un second souhait serait de disposer d'outils qui génèrent des allocations qui seraient conformes aux exigences de sécurité du système étudié.

Notre objectif principal est d'assister les analyses CCA par l'utilisation des techniques de méthodes formelles. Nous aurons donc deux sous-objectifs conformes aux préoccupations des ingénieurs :
- proposer un moyen pour vérifier automatiquement qu'une allocation est sûre.
- proposer un moyen pour générer automatiquement des allocations sûres.

Pour le premier sous-objectif, notre démarche consiste à modéliser à l'aide d'un langage formel (AltaRica dans notre cas), les architectures fonctionnelles et matérielles du système ainsi que l'allocation des fonctions sur les composants de l'architecture matérielle. Nous construisons deux modèles : le modèle fonctionnel qui est restreint à l'architecture fonctionnelle et un modèle global qui décrit également l'architecture matérielle et l'allocation. Il faut également formaliser les exigences de sécurité que le système doit vérifier. Il est ensuite possible de vérifier automatiquement la tenue de ces exigences pour les deux modèles construits. Finalement, nous proposons de comparer les résultats des vérifications pour les deux modèles de façon à déterminer si l'impact de l'allocation est acceptable du point de vue de la sécurité.

Pour le second sous-objectif, nous proposons d'utiliser des techniques de résolution de contraintes. Il faut donc modéliser le problème de l'allocation des éléments de l'architecture fonctionnelle sur les éléments de l'architecture matérielle sous forme de contraintes. Il est aussi nécessaire de décrire des directives d'allocation telles que la ségrégation entre fonctions. Ces directives peuvent être émises par les concepteurs des systèmes mais nous proposons aussi une technique permettant de dériver ces directives à partir des résultats des analyses de sécurité effectuées sur la base du modèle fonctionnel. Finalement, nous avons étudié la possibilité de suggérer des modifications de l'architecture matérielle lorsqu'aucune allocation ne peut être proposée par les techniques de résolution de contraintes.

1.5 Plan de lecture de la thèse

Dans le chapitre 2, nous commençons par introduire les concepts de modélisation de systèmes et de vérification formelle de propriétés. Nous présentons ensuite, relativement au cadre d'étude de la sûreté de fonctionnement, le type d'exigences que l'on souhaite vérifier ainsi qu'un survol des langages de modélisation adaptés. Nous présentons pour finir ce chapitre le langage de modélisation AltaRica que nous avons choisi pour nos travaux ainsi que les différents types d'analyses associés.

Ensuite, dans le chapitre 3, nous décrivons notre approche pour évaluer l'impact d'une allocation sur la sécurité d'un système. Nous montrons comment modéliser les architectures fonctionnelles et matérielles ainsi que les relations d'allocation qui les relient. Nous illustrons toutes les notions introduites dans ce chapitre sur l'exemple simple de l'architecture COMmand/MONitoring.

Dans le chapitre 4, nous présentons une méthode de génération automatique d'allocations. Nous montrons comment modéliser sous forme de contraintes les architectures fonctionnelles, matérielles et les directives d'allocation. Les différents types de directives d'allocation considérées sont présentés dans ce chapitre. A nouveau, l'architecture COM/MON est utilisée pour illustrer l'approche proposée.

Dans le chapitre 5, nous montrons comment dériver les directives d'allocation à partir de résultats d'analyse obtenus sur la base d'un modèle de l'architecture fonctionnelle. Nous étudions également le cas où il n'est pas possible de trouver une allocation sûre et nous montrons comment modifier l'architecture matérielle de façon à pouvoir trouver des allocations sûres.

Dans le chapitre 6, nous appliquons l'approche présentée dans les chapitres précédents au système de Suivi de Terrain d'un avion de chasse. Nous illustrons l'ensemble des techniques proposées de façon à rechercher une allocation sûre des fonctions de ce système sur une architecture matérielle informatique.

Dans le chapitre 7, nous validons l'approche proposée à une étude de cas de taille industrielle : un système de génération et de distribution hydraulique d'un avion civil. Dans ce chapitre, nous validons plus particulièrement les techniques de modélisation et de vérification d'allocation dans le cas particulier de l'installation des équipements d'un système au sein d'un avion.

Enfin, dans le chapitre de conclusion, après un bilan des travaux effectués dans cette thèse, nous comparons notre approche avec les travaux similaires existants. Nous terminons en examinant quelques perspectives ouvertes par ce travail.

Chapitre 2

Techniques pour la modélisation et la vérification

Nous commençons par présenter l'intérêt de la modélisation et de la vérification formelle de propriétés sur des modèles, avant de détailler chacune des étapes nécessaires : la modélisation, la formalisation des propriétés à vérifier, et enfin la vérification de ces propriétés sur le modèle. Nous déclinons ensuite chacune de ces étapes dans notre domaine de la vérification de propriétés de sûreté de fonctionnement. Nous formalisons le type de propriété qui nous intéressent, puis abordons les principaux langages de modélisation utilisés pour faire des analyses de sûreté de fonctionnement, avant de nous focaliser sur la définition du langage AltaRica, que nous avons choisi d'utiliser dans nos travaux. Nous terminons en listant les outils d'analyse existants, permettant de manipuler des modèles AltaRica.

Sommaire

2.1 Introduction à la modélisation et à la vérification de systèmes

Pour vérifier qu'un système présente un comportement attendu, une première approche consiste à construire ce système, puis à observer son comportement face à différents stimuli (tests du système). Ce type d'approche présente deux inconvénients majeurs :

- Tout d'abord, il nécessite de disposer du système réel, et donc d'être en fin du cycle de développement du système. Par conséquent, en cas de test non conforme aux attentes, des travaux très couteux de « *rework* » sur le système peuvent être nécessaires.
- Ensuite, ce type d'approche par test ne permet pas d'obtenir de garantie absolue sur le système car

  1. il est impossible d'obtenir une couverture totale d'un système dès qu'il atteint un niveau de complexité important, et
  2. sachant qu'il est très peu probable d'observer des événements pouvant impacter le comportement du système (panne matérielle, permutation de bit résultant d'un champ électromagnétique, etc.), il est très improbable de pouvoir observer leur impact sur le système par test.

Finalement, dans le cas de la conception de systèmes ayant des fonctions critiques, cette approche de validation par tests s'avère insuffisante. Il est nécessaire de pouvoir, dès la conception d'un système critique, orienter les choix de design, et vérifier le plus tôt possible, que les choix de conception du système sont cohérents avec les objectifs visés.

Pour ce faire, une première étape de modélisation du système est nécessaire. Le but est de représenter par un modèle une vision abstraite de ce que sera notre système une fois finalisé. Il faut comprendre par « vision abstraite » que seules les informations ayant un impact sur le comportement que l'on souhaite vérifier sont prises en compte.

2.1.1 Modélisation d'un système

Pour modéliser un système, il est pratique de le décomposer en un ensemble de composants. Cette décomposition est assez naturelle car un système remplit une ou plusieurs fonctions en utilisant un ensemble de sous-systèmes qui interagissent. Par exemple, dans un système hydraulique d'avion, la puissance fournie repose sur un ensemble de composants qui sont des tuyaux, des pompes, des vannes qui interagissent entre-eux.

Il est donc nécessaire de modéliser en premier lieu chacun des composants constituant le système, en prenant en compte ses caractéristiques relatives à ce que l'on souhaite vérifier.

Une fois le comportement interne des composants modélisé, il est ensuite nécessaire de modéliser les interactions existantes entre ces composants.

Bien entendu, les critères que l'on prend en compte pour cette modélisation dépendent du type de vérification que l'on souhaite faire par la suite. Des langages d'expression de modèles ont été définis spécifiquement aux différents critères à observer. Nous en énumérons quelques uns pour donner un aperçu de la diversité du monde de la modélisation des systèmes[1].

- la représentation de l'architecture d'un système avec les langages de modélisation UML [EC97] et AADL [FGH06] sous forme d'un ensemble de composants interagissant. Le système sera décrit essentiellement du point de vue de sa structure : les caractéristiques intra-composant ainsi que les caractéristiques inter-composants (les interactions).
- la représentation d'un système dans l'espace en trois dimensions avec IRIS [Air00] ou CATIA [DAS85],
- la représentation des comportements possibles d'un système à l'aide de formalismes dédiés :
  - les automates à contraintes, utilisés par le langage AltaRica [APGR00], dans lesquels les contraintes régissent le franchissement des transitions,

[1]Nous reviendrons de manière plus détaillée sur les types de modèles correspondant à nos besoins de modélisation en section 2.2.2

– les automates temporisés, utilisés notamment par le langage Uppaal[PL00],
– les réseaux de Petri [Pet81] permettant de manière très simple, de modéliser les interactions entre composants du type *provider* fournissant une donnée, un service auprès d'un *receiver* en attente de cette fourniture pour à son tour accomplir sa fonction.

Pour chacun de ces types de modèles, des attributs spécifiques sont définis et caractérisent le système à modéliser :
– pour les modèles AADL, un ensemble de composants avec identificateurs, attributs et services spécifiques doivent être identifiés, ainsi que les relations existantes entre ces composants,
– pour les modèles en trois dimensions, chacun des composants du système à modéliser doit être représenté en trois dimensions, ainsi que chacune des connexions entre composants,
– pour les automates à contraintes, il s'agit de caractériser l'ensemble des états d'un automate, les transitions existantes entre ces états, et les contraintes de franchissement associées,
– pour ce qui est des réseaux de Petri, il s'agit de définir l'ensemble des places du réseau ainsi que toutes les transitions, puis enfin le marquage initial des places.

La vérification d'une propriété sur un système modélisé ne peut bien sûr porter que sur les informations contenues dans le modèle au travers des éléments qui le composent, et sur la dynamique du système modélisé. Il est donc important de connaître tout d'abord le type de propriétés que l'on souhaite vérifier sur un système avant de pouvoir choisir un moyen de modélisation permettant de représenter toutes les caractéristiques du système impactant la satisfaction de la propriété.

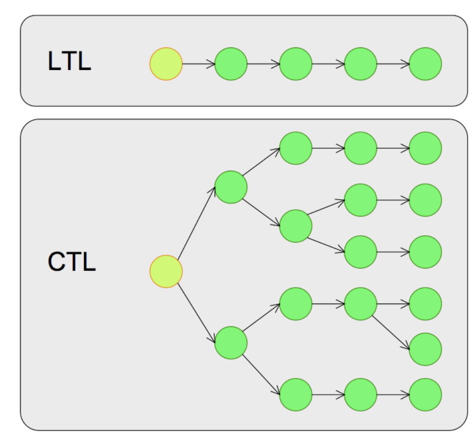
2.1.2 Expression des exigences que le modèle doit vérifier

Définir une exigence (ou propriété) qu'un système doit satisfaire consiste tout d'abord à exprimer en langage naturel cette exigence. Il peut s'agir par exemple, dans le cas d'un système hydraulique d'avion, de l'exigence suivante : « *le système hydraulique doit toujours fournir une puissance suffisante à ses consommateurs identifiés comme critiques* » .

Cette exigence définie en langage naturel doit être formalisée et précisée pour pouvoir être vérifiée de manière automatique sur un modèle. Il s'agit d'exprimer l'exigence de manière logique. Il est tout d'abord nécessaire de définir l'ensemble des éléments à considérer. Dans notre exemple, il faut identifier et modéliser le *système hydraulique*, puis l'ensemble des *consommateurs critiques* de la puissance qu'il fournit (les gouvernes, etc.), ainsi que la *puissance* minimale (suffisante) pour chacun des consommateurs. Une fois ces éléments caractérisés, il est possible de formaliser une partie de l'exigence qui est :
P = *Le système hydraulique fournit une puissance suffisante aux consommateurs critiques.*

Pour exprimer formellement qu'une propriété doit être vérifiée par tous ou certains des états accessibles d'un système, il est nécessaire de manipuler des ensembles d'états, pour vérifier que chaque état dans cet ensemble satisfait la propriété. Les logiques temporelles permettent de manipuler ce type d'ensembles. Parmi elles, nous en citerons trois :

LTL : *Linear Temporal Logic* [Pnu77], permettant d'exprimer des propriétés devant être satisfaites par les différents états accessibles d'un système au cours d'une évolution du système donnée (sur un chemin donné représentant les états successifs du système, comme illustré par la figure ci-contre. Par exemple, on peut exprimer les propriétés du type « dès que le système tombe en panne, une alarme doit le signaler ». Plus généralement, les propriétés peuvent utiliser les expressions suivantes pour exprimer une propriété sur les états successifs d'un système : *immédiatement après, un jour, toujours, jusqu'à.*



CTL : *Computational Tree Logic* [EH82], permet de considérer plusieurs évolutions possibles à partir d'un état donné du système plus tôt que d'avoir une vue linéaire du système considéré. Ainsi, depuis un état du système, on peut exprimer une propriété du type « le système ne doit jamais (par quelque évolution du comportement du système possible) être dans un état de panne non signalé »

µcalcul : le *µcalcul*[EL86] est la logique présentant le plus grand pouvoir d'expressivité. En effet, elle dispose des opérateurs ν « plus grand point fixe » et μ « plus petit point fixe ». La notion de point fixe s'applique à des ensembles d'états. Par exemple, on utilise le plus petit point fixe pour calculer un ensemble d'états accessibles depuis un état donné. Le point fixe est atteint lorsque depuis le dernier ensemble d'états calculé, aucun nouvel état n'est accessible. Le plus grand point fixe peut être utilisé pour identifier un ensemble d'états ne menant pas à un état bloquant. Les opérateurs de point-fixe peuvent être utilisés afin d'exprimer les opérateurs *Toujours, un jour, etc.* des logiques CTL et LTL.

L'expression de la propriété P portant sur un état du modèle doit quant à elle être basée uniquement sur des informations définies par les éléments du modèle. Dans le cas d'un automate, il peut s'agir d'une propriété du type *l'ensemble des états représentant des situations non désirées de l'automate n'est pas atteignable* et dans le cas de modèles en trois dimensions, il peut s'agir de propriété du type : *Le composant X et le composant Y sont distants d'au moins 20 cm.*

Ensuite, il est nécessaire de préciser dans quels cas cette propriété doit être satisfaite. Dans l'expression en langage naturel de l'exemple de propriété, le mot « *toujours* » peut aussi être interprété par des valeurs probabilistes :

- probabilité (P satisfaite) $= 1$ ou encore
- probabilité (P satisfaite) $\geq 1 - 10^{-9}$ si une très faible probabilité (inférieure à 10^{-9}) de non respect de la propriété est jugée acceptable.

2.1.3 Vérification automatique par exploration du modèle(*Model Checking*)

Une fois construit le modèle du système étudié et formalisées les propriétés, il reste à vérifier que le modèle satisfait les propriétés. Pour cela, nous nous sommes intéressés aux techniques fondées sur l'exploration automatique du modèle, le « Model-Checking ». Ces outils parcourent l'espace d'états correspondant à toutes les évolutions du modèle du système et évaluent les propriétés dans chacun des états atteints. L'évaluation des propriétés peut être réalisée sur la base d'un espace d'états complet et calculé au préalable, soit en parcourant progressivement l'espace d'état, ceci permet ne pas parcourir les états qui ne sont pas pertinents pour une propriété, soit en se fondant sur une représentation symbolique de l'espace d'état, ceci permet de parcourir et évaluer en une seule étape un ensemble d'états atteints. Dans ce dernier cas, la représentation interne des états accessibles et le parcours de ces derniers pour la vérification peuvent être réalisés à l'aide de BDD[2][Bry86]. Une alternative consiste à représenter l'espace d'états comme une formule booléenne puis à utiliser des techniques de

[2]Binary Decision Diagram

satisfaction de contraintes booléennes pour vérifier les propriétés.

En pratique, les outils de vérification automatique de propriétés sont spécifiques à :
– un type de modèle ou un langage de modélisation particulier,
– et un type de propriétés, exprimées au moyen d'une logique et de prédicats donnés.

Parmi les outils de *ModelChecking* existants, nous citerons tout d'abord MEC5 [Vin03, ABC94] permettant de vérifier des propriétés exprimées par des formules de μ-calcul [EL86] sur des modèles AltaRica. Ensuite, les outil de la famille SMV (Carnegie Mellon SMV, Cadence Labs SMV et NuSMV) [CCGR00] vérifient sur des modèles du type structure de Kripke [McM98] des propriétés exprimées avec des formules CTL ou LTL. L'outil TINA [BV06], quant-à-lui, manipule des réseaux de Petri sur lesquels il vérifie des propriétés exprimées par des formules LTL, CTL et TCTL[Yam95]. UPPAAL [PL00] est un langage de modélisation mais aussi un outil permettant de vérifier sur des modèles de type automates temporisés, des propriétés d'invariants et d'atteignabilité par l'exploration de l'espace d'état symbolique du système représenté par des contraintes.

Bien entendu, nous n'envisageons pas de vérifier toutes les exigences par le biais du model-checking. En effet, les exigences relatives au placement des équipements dans un avion comme (« *Les équipements doivent être distants d'au moins L mètres* ») ou (« *Les équipements électriques doivent se trouver au dessus des équipements hydrauliques* ») seront vérifiées à l'aide de techniques propres aux outils de modélisation géométrique.

2.2 Modélisation et vérification de la sécurité des systèmes aéronautiques

Après avoir introduit dans la section précédente les concepts génériques de la modélisation et de la vérification des systèmes, nous les instancions avec les techniques utiles dans le domaine de la sécurité des systèmes embarqués aéronautiques. Nous commençons par définir le type d'exigences que l'on souhaite vérifier, avant de déduire le type de modèles que l'on doit manipuler. Nous présentons dans ce cadre les types de modèles utilisés jusqu'alors pour effectuer des analyses de sécurité sur des systèmes industriels. Nous justifions ensuite notre choix du langage de modélisation AltaRica avant de présenter les analyses que proposent les outils manipulant ce type de modèle.

2.2.1 Exigences de sûreté à vérifier

Nous nous intéressons aux exigences de sûreté de fonctionnement et plus particulièrement aux exigences de sécurité-innocuité (au sens « *safety* ») qui visent à éviter les défaillances catastrophiques, c'est-à-dire, selon [Lap96], celles pour lesquelles des conséquences sont inacceptables compte tenu du risque encouru par les personnes ou les biens.

Les exigences qualitatives de sécurité auxquelles nous nous intéressons sont de la forme : « *En présence de X pannes, le système n'atteint jamais un certain état redouté* ». Lorsque les spécifications du système le permettent, il est possible d'ajouter des probabilités d'occurrence aux défaillances permettant ainsi la vérification d'exigences de type quantitatif. Elles sont généralement de la forme : « *La probabilité que le système soit dans un état redouté doit être inférieure à* 10^{-X} ».

Les états redoutés que nous souhaitons éviter correspondent aux *Failure Condition* décrites dans l'ARP [47696]. Ces *FCs* sont classées suivant leur niveau de criticité. Ceci permet de définir les exigences quantitatives et qualitatives à vérifier. En effet, suivant le degrés de criticité d'une situation, il faut respecter certaines exigences. Par exemple, lorsqu'une situation est qualifiée « catastrophique », alors il faut au minimum des combinaisons de 3 défaillances pour y parvenir. De même, la probabilité associée à la présence d'une telle situation est de 10^{-9} par heure de vol. La correspondance entre les exigences qualitatives (portant sur le nombre de pannes) et quantitatives (portant sur la probabilité d'occurrence d'une situation) suivant la classification des situations est décrite par le tableau suivant :

Conséquences	Probabilités de pannes $> 10^{-3}$	$< 10^{-5}$	$< 10^{-7}$	$< 10^{-9}$
Mineure				
Significative	interdit			
Critique	interdit	interdit		
Catastrophique	interdit	interdit	interdit	
Exigence Qualitative	Simple	Double	Double+	Triple
	Nombre de pannes			

Lorsque le système étudié a un comportement dynamique lié, par exemple, à des actions de reconfiguration, il faut habituellement examiner non pas un état mais une séquence d'états pour déterminer si la situation redoutée est atteinte. Dans ce cas, nous avons recours aux opérateurs de la logique temporelle pour formaliser la situation redoutée.

2.2.2 Langages de modélisation

Les principaux langages de modélisation utilisés dans la sûreté de fonctionnement sont des langages permettant d'observer l'impact de la défaillance de l'un de ses composants sur l'ensemble du système afin de déterminer si cela conduit à une des situations redoutées identifiées.

Parmi les différents langages de modélisation existants, nous avons choisi de ne présenter que les langages qui ont été utilisés pour la vérification de propriétés de sécurité. Nous ferons donc une brève présentation des langages AADL pour la description des systèmes, des réseaux de Petri pour la simulation des systèmes dynamiques, du langage associé au model-checker SMV et enfin du langage AltaRica qui permet la génération automatique d'arbres de défaillances.

Le Langage AADL

AADL [3] [FLV03, FGH06, Lew06] a été conçu pour permettre la conception et l'analyse de systèmes complexes, critiques, temps réel dans les domaines de l'avionique, l'automobile et le spatial. Ce langage est en cours de standardisation sous l'autorité du SAE[4] dont une première version stable AADL 1.0 a été publié en novembre 2004.

Langage de description Le langage AADL V1 permet de décrire comment les composants sont combinés en sous-systèmes et comment ils interagissent entre eux. Les architectures sont décrites de manière hiérarchique. Les composants sont les « briques de base » des architectures AADL . Ils sont groupés en trois catégories :

1. **logicielle :** processus, sous-programme, donnée, fil d'exécution « thread », groupe de fils d'exécution.
2. **plate-forme :** processeur, mémoire, dispositif, bus.
3. **composite :** système.

A chacune de ses catégories, la standardisation du langage associe une icône permettant une représentation normalisée des différents composants d'un système.

[3] Architecture Analysis and Design Language
[4] Society of Automotive Engineers

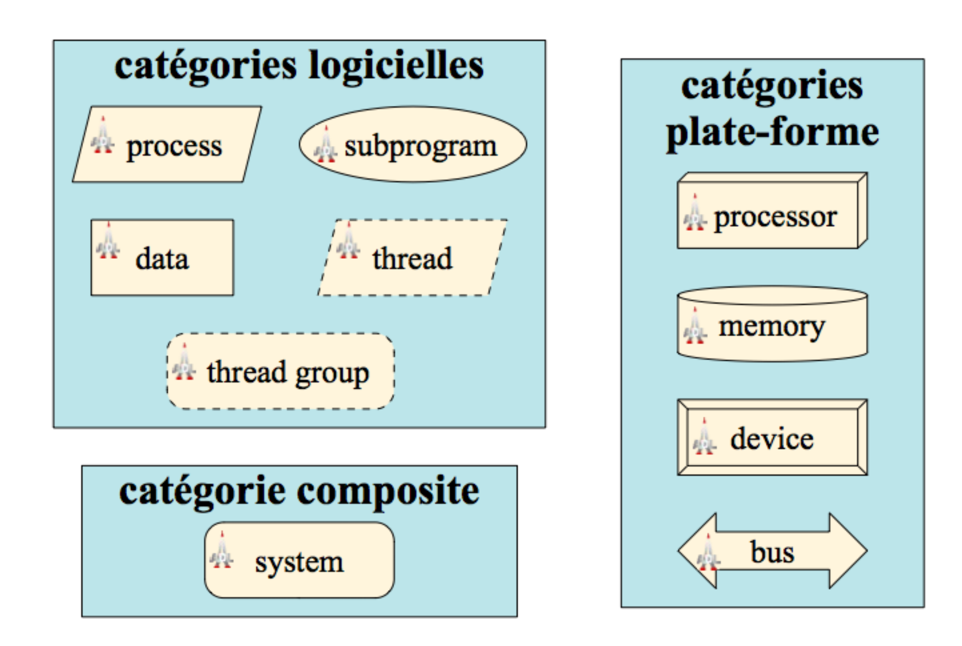


Fig. 2.1 – représentation graphique des composants AADL

Chaque composant AADL a deux niveaux de description : le type et l'implémentation. Le type (`component types`) correspond à l'interface fonctionnelle du composant, il décrit comment le composant est « visible » par l'environnement (c-à-d. quelles sont ses propriétés et caractéristiques). L'implémentation (`component implementations`) correspond à la structure du composant en terme de sous composants, connexions et modes opérationnels (plusieurs implémentations peuvent être associées à un même type).

À chaque composant on peut associer des propriétés et leur donner des valeurs. Celles-ci permettent de caractériser le composant. Certaines propriétés sont prédéfinies, elles sont identifiées par un nom, un type et la liste des catégories de composants sur lesquelles elles s'appliquent. Par exemple les tâches (`thread`) disposent de propriétés temps-réel telles que la période, l'échéance ou la durée d'exécution.

L'interconnexion entre les composants est permise grâce à différents ports sur les composants. Un port est un point d'entrée et/ou de sortie d'un composant, par où peuvent transiter des données (`data`), des événements (`event`) ou même des événements associés à des données (`data event`). Une connexion permet de relier deux ports, soit les ports de deux sous-composants, soit le port d'un sous-composant avec le port du composant le contenant.

Les événements peuvent déclencher un changement de mode de comportement d'un composant. En effet, les modes (`modes`) permettent de modéliser la reconfiguration du système. Ces changements de modes permettent de représenter des architectures dynamiques (certains composants peuvent être activés ou désactivés en fonction du mode).

Le langage AADL est conçu pour décrire des architectures statiques avec des modes opérationnels pour leurs composants. Néanmoins, le langage de base peut être étendu afin de permettre à l'utilisateur d'ajouter des informations supplémentaires à l'architecture. Les *modèles d'erreur* AADL sont une extension qui a pour objectif de permettre la réalisation d'analyses (qualitatives et quantitatives) de sûreté de fonctionnement. Le « AADL Error Model Annex » [AW05] définit un sous langage qui permet la déclaration de modèles d'erreur qui seront instanciés et associés à chaque composant du système.

Ce langage permet d'analyser l'impact des choix architecturaux sur les exigences du système [Rug05].

Réseaux de Petri P/T

Un réseau de Petri Places-transitions est un moyen de modéliser le comportement des systèmes dynamiques à événements discrets. Il se base sur une description des relations existantes entre des conditions et des événements.

Un réseau de Petri est défini par un graphe orienté comportant (cf. figure 2.2) :
- un ensemble fini de places, P={P1, P2, P3, ..., Pm}, symbolisées par des cercles et représentant des conditions :
  - une ressource du système (ex. : une machine, un stock, un convoyeur,...)
  - l'état d'une ressource du système (ex. : machine libre, stock vide, convoyeur en panne,...)
- un ensemble fini de transitions, T={T_1, T_2, T_3, ..., T_n}, symbolisées par des tirets et représentant l'ensemble des événements (les actions se déroulant dans le système) dont l'occurrence provoque la modification de l'état du système,

Place : Transition :

- un ensemble fini d'arcs orientés qui assurent la liaison d'une place vers une transition ou d'une transition vers une place,

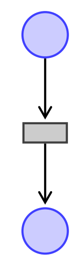


Fig. 2.2 – Exemple de Reseaux de Pétri P/T

Franchissement d'une transition Une transition est franchissable lorsque toutes les places qui lui sont en amont (ou toutes les places d'entrée de la transition) contiennent au moins un jeton (•). Le franchissement consiste à retirer un jeton de chacune des places d'entrée et à rajouter un jeton à chacune des places de sortie de la même transition (voir exemple figure 2.3).

Extensions Plusieurs extensions du langage permettent d'utiliser les réseaux de Petri sur des problèmes industriels comme, les réseaux de Petri colorés et hiérarchiques (des éléments du réseau de Petri sont eux-mêmes composés d'un réseau de Petri) [LM88]. Dans un réseau de Petri coloré, on associe une valeur à chaque jeton (contrairement au réseau de Petri de base ou il n'existe qu'une sorte de jeton). Cette distinction entre les jetons permet d'observer la propagation d'une panne dans un système par le déplacement d'un jeton particulier dans les places représentant les composants impactés du système.

Une autre extension intéressante concerne les réseaux de Petri Stochastiques. Dans ces réseaux, on associe à chacune des transitions une variable aléatoire représentant une condition pour le tir de cette transition. Cette variable correspond au taux de défaillance du franchissement où un nombre aléatoire va "forcer" arbitrairement le choix du chemin à suivre.

La simulation du système par ce réseau va permettre de calculer la probabilité d'occurrence d'un événement. Pour effectuer ce calcul, une simulation va comptabiliser le nombre de passage par une certaine place choisie et ce nombre va ensuite être divisé par le nombre total de simulation effectué. Nous obtiendrons ainsi la probabilité que l'événement défini par la place, se réalise ou non.

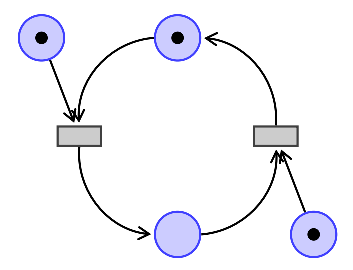


Fig. 2.3 – Exemple de franchissement de transitions d'un réseau de Petri P/T

SMV

Le langage SMV[McM98] est un langage de description à l'origine conçu pour la vérification de systèmes matériels[BCMD90]. Il permet de décrire les systèmes de manière modulaire et hiérarchique. Les systèmes manipulés sont décrits par des ensembles finis d'états.

Globalement, le langage SMV permet de décrire

- le modèle, à l'aide de déclarations de types, de signaux (séquences infinies d'un type donné), d'affectations, de modules (types structurés, etc), de conditions et de boucles (loop), etc.
- les propriétés que le modèle doit satisfaire, formalisées à l'aide d'expressions logiques, d'assertions, exprimées avec la logique CTL ou LTL [EH82].

Le formalisme du langage SMV est proche de celui d'AltaRica pour les types communs. Une passerelle permettant de transformer des modèles AltaRica vers des modèles SMV existe et permet de bénéficier des outils d'analyse de modèles SMV aux modèles AltaRica. Voyons à présent les différents types d'analyses de système qui sont disponibles sur des modèles SMV.

Tout d'abord, il est possible d'effectuer des vérifications formelles compositionnelles de propriétés par *preuve*. Le principe de la vérification compositionnelle est assez simple : il consiste, lorsqu'une vérification d'une propriété sur le système global est jugée trop compliquée, à la décomposer en vérification sur les sous-systèmes moins complexes qui la composent.

De plus, afin de vérifier une propriété, il est possible d'avoir recours à des hypothèses (assumptions) exprimées par le mot-clef `assume`. Cela peut grandement faciliter une preuve, mais cela demeure un mécanisme dangereux s'il est utilisé à tort ce qui est le cas, par exemple, lorsque les hypothèses ne sont pas cohérentes entre elles.

Un autre type d'analyse est la vérification du *raffinement* d'un modèle. Il y a raffinement d'un modèle lorsque depuis un premier modèle abstrait décrivant un système, un second modèle plus détaillé est dérivé, décrivant le même système. La difficulté réside dans la cohérence entre les différents modèles du même système, et dans la préservation des bonnes propriétés. Pour garantir la cohérence entre plusieurs modèles décrivant un même système, le langage SMV propose d'utiliser un *refinement maps*, appelé « invariant de collage » en langage B [Cle07]. Il permet de décrire la correspondance entre les propriétés exprimées sur les différents modèles concernés.

Langage AltaRica

AltaRica est un langage créé par le Laboratoire Bordelais de Recherche en Informatique (LaBRI) au milieu des années 90 [APGR00, Poi00, PR99]. Ce langage formel est né de la volonté d'industriels et de chercheurs d'établir divers ponts entre la sûreté de fonctionnement et les méthodes formelles.

Nous avons choisi ce langage pour notre démarche pour l'ensemble des raisons suivantes :

- AltaRica est basé sur les automates à contraintes [FP93] et spécifie un système comme un ensemble de variables contraintes par des formules. L'évolution de ces variables est définie par les étiquettes des transitions. Il permet ainsi de combiner la description d'un comportement

fonctionnel et dysfonctionnel d'un système à l'aide de contraintes sur les différentes variables caractérisants ses composants.

- La représentation dysfonctionnelle du comportement d'un système est simplifiée grâce à l'introduction de transitions modélisant la défaillance des composants.
- Ce langage permet d'effectuer des analyses quantitatives ou qualitatives à partir du même modèle selon que l'on prend en compte les lois de probabilité associées aux transitions.
- Sa capacité à réaliser des modèles compositionnels et hiérarchiques lui permet de modéliser des systèmes complexes.
- Les modèles AltaRica peuvent être édités de manière graphique par différents outils fournis par les industriels partenaires du projet et ils peuvent aussi être édités et manipulés directement par le formalisme de définition du langage.

Les principes de l'atelier AltaRica sont résumés dans la figure 2.4. Un modèle AltaRica permet de décrire des systèmes réels et peut être traduit dans plusieurs formalismes de plus bas niveau permettant différents types d'analyses (comme illustré par la figure 2.4).

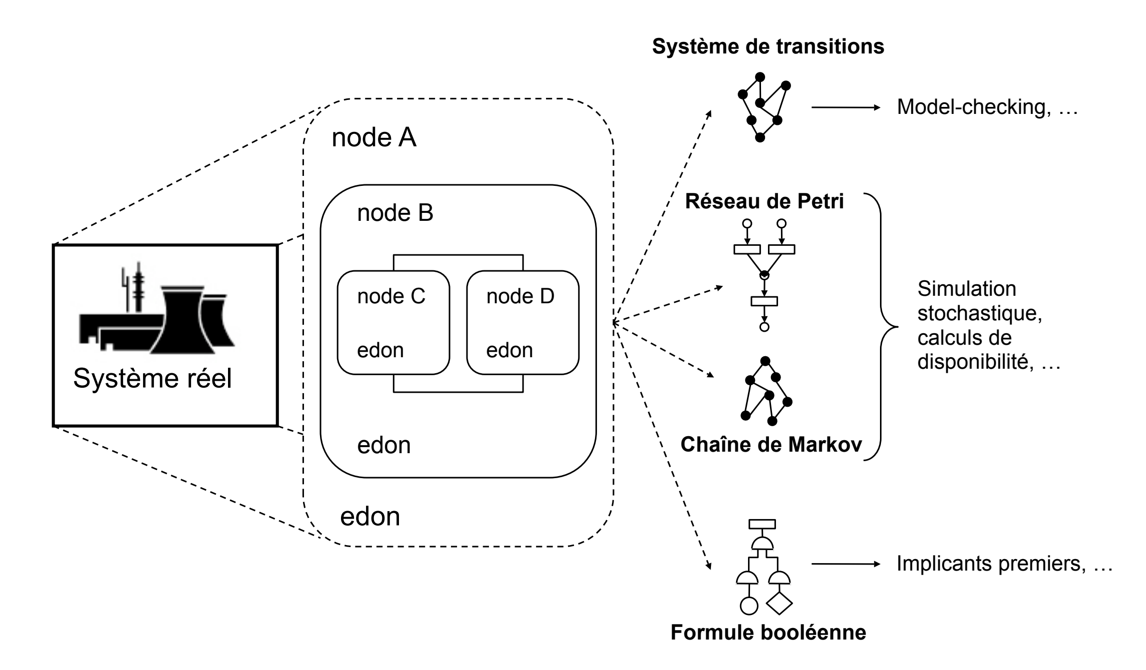


Fig. 2.4 – Atelier AltaRica

Nous présentons ci-après plus en détails ce langage et les outils qui lui sont associés.

Les industriels intéressés par le langage AltaRica, Dassault System et EADS APSYS ont adopté ce langage pour leurs outils de modélisation qui sont respectivement Cecilia OCAS et SIMFIA. Le langage AltaRica utilisé par ces outils est une restriction du langage d'origine en vue de faciliter la compilation. Le nombre de variables autorisées est fini, les variables d'état appartiennent à des intervalles finis d'entiers et les flux ne sont plus bidirectionnels. On a également une simplification de la notion de priorité entre événements.

Dans cette thèse, nous nous intéresserons à la version Data Flow utilisée au sein de l'outil OCAS que nous avons utilisé pour réaliser nos modèles et certaines analyses.

Cette version allégée du langage est également utilisée par la ToolBox d'Arboost Technologies (aussi appelé Combava) d'Antoine Rauzy [Rau]. Cette ToolBox rassemble plusieurs outils communs à OCAS et SIMFIA et permet en outre de faire du model-checking.

L'IRCCyN a aussi développé une extension temps réel d'AltaRica [Pag04, CPR04] basée sur les automates à contraintes temporisés qui se compilent en automates temporisés. L'outil Tarc permet de réaliser cette compilation.

Enfin, le LaBRI développe des outils basés sur le langage d'origine. Le model-checkeur MEC V

utilise le formalisme des BDD afin d'optimiser la vérification de formules de μ-calcul et il est maintenant intégré dans ARC[LaB07]. L'utilisation du model-checking permet de s'assurer du parcours complet du graphe des états accessibles.

2.2.3 Présentation détaillée du langage choisi : AltaRica Data-Flow

Concepts

Pour la modélisation du système en AltaRica, certaines règles de conception de composants sont définies. Les concepts utilisés ci-après sont illustrés sur un exemple trivial de modèle représentant le comportement d'un `Capteur`.

- Un composant modélisé est représenté par un nœud (`node`) caractérisé par son nom (*Sensor* dans notre exemple : figure 2.5).
- Un noeud possède au moins une variable d'état (`state`) dont la valeur dénote le fonctionnement (ou dysfonctionnement) du composant. La valeur de cette variable à l'état initial est spécifiée dans la clause `init` (dans notre exemple, une variable booléenne suffit pour représenter la présence ou l'absence d'une défaillance).
- Il possède également un nombre fini de variables de flux (`flow`) permettant de représenter les connexions possibles (`in`/`out`) entre les composants. Le type de ces variables représente le niveau d'abstraction des valeurs échangées (dans l'exemple, nous considérons que le composant fournit en sa sortie *capt_out* une valeur dénotant son état interne).
- Plusieurs événements (`event`) sont nécessaires pour modéliser les défaillances ou les reconfigurations qui provoquent les changements d'états décrits dans les transitions (dans l'exemple, le composant est sujet à une défaillance *fail_error*).
- Les transitions (`trans`) représentent les aspects temporels des modes de fonctionnement et de dysfonctionnement. Les gardes de ces transitions représentent la cause des événements. La partie droite de la transition permet l'affectation à considérer lorsque la transition est tirée (dans notre exemple, la transition a pour effet le changement de la valeur de la variable d'état).
- La partie assertion d'un composant (`assert`) regroupe les relations de dépendance entre l'état du composant et la valeur de ses flux permettant ainsi de restreindre et de supprimer les comportements impossibles ou inutiles (dans l'exemple, nous souhaitons limiter la valeur de la variable de sortie à la variable d'état).
- L'état initial du composant (`init`) déclare la situation nominale que le système doit aborder.
- Des informations, comme les probabilités d'occurrence associées à des événements, peuvent être ajoutées aux différents composants du système dans la partie `extern`. Ces informations peuvent ensuite être utilisées par des outils de traitement de coupes minimales comme Aralia, par exemple.

La création d'un modèle AltaRica passe tout d'abord par l'application de ces règles de conception pour chacun des noeuds de l'architecture fonctionnelle. Les connexions des différents noeud de cette architecture sont ensuite transformées en flux de données entre les composants AltaRica correspondants pour terminer la création du modèle AltaRica.

Le code AltaRica de notre exemple et la sémantique qui lui est associée sont représentés par la figure 2.5.

```
node Sensor
 state
  //boolean for the state variable
  Status : bool;
 flow
  //boolean for the output of the Captor
  capt_out : bool : out ;
 event
  //failure
  fail_error;
 trans
  //failure effect
  Status |- fail_error -> Status = false
 assert
  //behavior of the Captor
  capt_out = Status;
edon
```

fail_error
Status = true
Status = false

FIG. 2.5 – Code Altarica et automate de comportement d'un Capteur

La définition formelle d'un modèle AltaRica est basée sur un automate de mode. Commençons par rappeler la définition d'un automate de mode :

Définition 2.1 : [Rau02]
Formellement, un automate de mode $\mathcal{M}$ *est un 9-uplet* $\mathcal{M} = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I\rangle$ *avec :*
- *D est un domaine fini*
- *Notons V l'ensemble fini de variables partitionné en trois classes S, F^{in} et F^{out}. Ces sous-ensembles de V sont dénommés respectivement variables d'état, de flux d'entrée et de flux de sortie.*
- *$dom : V \rightarrow 2^D$ telle que $\forall v \in V, dom(v) \neq \emptyset$ associe à une variable son domaine*
- *Σ est un ensemble fini d'événements*
- *δ est une fonction partielle appelée* transition *: $dom(S) \times dom(F^{in}) \times \Sigma \rightarrow dom(S)$, $dom(S) \times dom(F^{in})$ est la garde de la transition. Cette fonction calcule la prochaine valuation des variables d'état considérées en fonction des valeurs courantes des variables d'état, des variables de flux d'entrée et de l'occurrence de l'événement provoquant le changement d'état.*
- *σ est une fonction totale appelée* assertion *: $dom(S) \times dom(F^{in}) \rightarrow dom(F^{out})$. Elle calcule la valeur des variables de sorties en fonction des valeurs des variables d'état et de flux d'entrée.*
- *$I \in dom(S)$ est une fonction partielle qui décrit les conditions initiales*

L'automate de mode correspondant à l'exemple illustré par la figure 2.5 est le suivant :

- $D = \{\texttt{true}, \texttt{false}\}$
- $S = \{\texttt{Status}\}$, $dom(\texttt{Status}) = \{\texttt{true}, \texttt{false}\}$
- $F^{in} = \emptyset$
- $F^{out} = \{\texttt{capt_out}\}$, $dom(\texttt{capt_out}) = \{\texttt{true}, \texttt{false}\}$
- $\Sigma = \{fail_error\}$
- $\delta(\texttt{true}, \bot, \texttt{fail_error}) = \texttt{false}$
- $\sigma(\texttt{true}, \bot) = \texttt{true}$
- $\sigma(\texttt{false}, \bot) = \texttt{false}$
- $I = \{(\texttt{true}, \texttt{true})\}$

Un système, tout comme son modèle, est un assemblage de composants élémentaires. Les composants peuvent être assemblés par connexion, regroupement ou bien par synchronisation. Voyons à

présent plus en détails ces différents moyens d'assemblage de composants définis par les automates de mode.

Hiérarchie asynchrone d'automates de mode

La composition d'automates de mode consiste à placer plusieurs automates de mode dans un même automate, une même vue, afin de pouvoir ensuite les faire communiquer et interagir (tant qu'ils n'interagissent pas entre eux, les noeuds caractérisés par un automate de mode sont considérés comme indépendants). Le résultat d'une composition d'automates de mode est un automate de mode regroupant les définitions de tous les automates de mode composés (cf. figure 2.6).

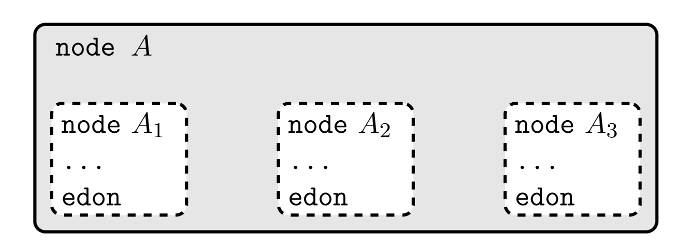


Fig. 2.6 – Composition d'automates de mode

La définition formelle de cette composition est la suivante :

Définition 2.2 :
Soient $A_1, ..., A_n$ n automates de mode, avec $A_i = \langle D, S_i, F_i^{in}, F_i^{out}, dom, \Sigma_i, \delta_i, \sigma_i, I_i \rangle$.

Supposons que leurs vocabulaires soient distincts :
- $\forall i, j, 1 \leq i \leq j \leq n, (S_i \cup F_i^{in} \cup F_i^{out}) \cap (S_j \cup F_j^{in} \cup F_j^{out}) = \emptyset$
- $\Sigma_i \cap \Sigma_j = \emptyset$

La composition parallèle de n automates $A_1, ..., A_n$ est un automate de mode noté :
$A = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I \rangle$ *tel que :*

- $S = \bigcup_{i=1}^{n} S_i$, $F^{in} = \bigcup_{i=1}^{n} F_i^{in}$, $F^{out} = \bigcup_{i=1}^{n} F_i^{out}$, $\Sigma = \bigcup_{i=1}^{n} \Sigma_i$
- *δ est obtenu en remontant les transitions δ_i de chaque automate A_i au niveau de A :*
  Soit $s_i \in dom(S_i)$ une valuation des variables d'état de A_i,
  Soient $i_i^{in} \in dom(F_i^{in})$ une valuation des entrées de A_i,
  Soient $t_i \in dom(S_i)$ et $e \in \Sigma_i$ tel que $t_i = \delta_i(S_i, I_i, e)$ alors :
  $$\delta(s_1, ..., s_n, i_1, ..., i_n, e) = \langle s_1, ..., s_{i-1}, t_i, s_{i+1}, ..., s_n \rangle$$
- *Les assertions sont définies de la même manière, par :*
  $$\sigma(s_1, ..., s_n, i_1, ..., i_n) = \langle \sigma_1(s_1, i_1), ..., \sigma_n(s_n, i_n) \rangle$$
- $I = \langle I_1, ..., I_n \rangle$

AltaRica est un langage hiérarchique : Un noeud intermédiaire est un noeud pouvant contenir d'autres noeuds (ces noeuds sont définis dans la clause `sub`). L'organisation complète d'un système est orchestrée autour d'un noeud principal appelé `Main` (le nom de ce noeud est une restriction imposée par l'outil OCAS). Finalement, la modélisation d'un système complet est représentée par un unique noeud AltaRica `Main` composé de différents noeuds représentant les éléments du système.

La connexion des interfaces d'automates de mode (synchronisme)

La connexion de composants, i.e. d'automates de modes, consiste à relier les composants par leurs interfaces pour leur permettre d'échanger des informations. Elle consiste donc à contraindre certaines variables d'entrée d'un composant à être égales à une sortie d'un autre composant.

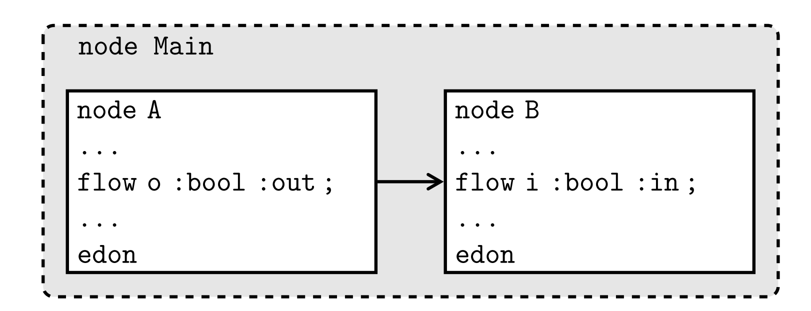


Fig. 2.7 – Connexion d'automates de mode

Pour que cette connexion soit valide, il est nécessaire que ces variables soient indépendantes, c'est à dire que les variables d'entrée n'interviennent pas dans les calculs de la valeur de la variable de sortie. Cette connexion doit être définie dans la rubrique `assert` du noeud père (Par exemple dans la figure 2.7, la connexion entre le noeud `A` et `B` est définie dans la partie assertion du noeud `Main`).

La définition formelle de la connexion d'automates est énoncée ci-après :

Définition 2.3 :

Soit $A_i = \langle D, S_i, F_i^{in}, F_i^{out}, dom, \Sigma_i, \delta_i, \sigma_i, I_i \rangle$ *un automate de mode,* $o \in F^{out}$ *et* $i_1, ..., i_k \in F^{in}$ *telles que* o *et* $i_1, ..., i_k$ *soient indépendants et* $dom(o) \subseteq dom(i_1), ..., dom(o) \subseteq dom(i_k)$.

Soient $F_\star^{in} = F^{in} \setminus \{i_1, ..., i_k\}$, $s \in dom(S)$, *et* $I \in dom(F_\star^{in})$.

Notons $I_{s,o/i_1,...,o/i_k}$ *la valuation de* F^{in} *telle que :*

$$I_{s,o/i_1,...,o/i_k}[u] \stackrel{def}{=} \begin{cases} I[u] & si\ u \in F_\star^{in} \\ \sigma(s, I')[o] & sinon \end{cases}$$

où $I' \in dom(F^{in})$ *est une extension quelconque de la valuation* $F_\star(F^{in} \sup F_\star^{in})$.

L'automate de mode A *où* $i_1, ..., i_k$ *sont connectés à* o *est l'automate de mode* $A_{o/i_1,...,o/i_n} = \langle D, S, F_\star^{in}, F^{out}, dom, \Sigma, \delta', \sigma', I \rangle$ *avec* δ' *et* σ' *tels que :*

- $\forall s \in dom(S), I \in dom(F_\star^{in})$ *et* $e \in \Sigma$ *si* $\delta(s, I_{s,o/i_1,...,o/i_n}, e)$ *est défini, alors* $\delta'(s, I, e) = \delta(s, I_{s,o/i_1,...,o/i_n}, e)$
- $\forall s \in dom(S), I \in dom(F_\star^{in}), \sigma'(s, I) = \sigma(S, I_{s,o/i_1,...,o/i_n})$

En pratique, une connexion peut être assimilée à un renommage des variables d'entrée par les variables de sortie.

La synchronisation d'automates

Les transitions dans les automates de mode sont asynchrones, i.e. par défaut, deux transitions ne peuvent être déclenchées simultanément. Seule l'opération de synchronisation [BR94] permet de tirer plusieurs transitions simultanément. L'ensemble des événements synchronisés est ainsi regroupé au sein d'un même vecteur de synchronisation. Ce nouveau vecteur de synchronisation introduit donc des connexions synchrones entre les noeuds synchronisés et est défini dans la rubrique `sync` du noeud père.

Avant de voir la définition formelle d'un vecteur de synchronisation pour un automate de mode, nous allons tout d'abord rappeler la notion de composition de jeux de valuations :

Soit $S \subseteq V$ (ensemble fini de variables de l'automate), $\mathcal{V}$, $\mathcal{V}'$ et $\mathcal{V}''$ des jeux de valuation des variables. On dit que $\mathcal{V}'$ et $\mathcal{V}''$ sont incompatibles par rapport à $\mathcal{V}$ si il existe une variable $v \in S$ telle que $\mathcal{V}[v]$, $\mathcal{V}'[v]$ et $\mathcal{V}''[v]$ sont distinctes. La composition des deux valuations $\mathcal{V}'$ et $\mathcal{V}''$ par rapport à $\mathcal{V}$, notée $\mathcal{V}' \circ_{\mathcal{V}} \mathcal{V}''$, est la valuation définie par :

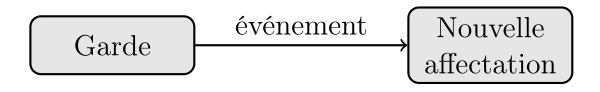
$$\forall v \in S,\ \mathcal{V}' \circ_{\mathcal{V}} \mathcal{V}'' \stackrel{def}{=} \begin{cases} \mathcal{V}'[v] & \text{si } \mathcal{V}'[v] \neq \mathcal{V}[v] \\ \mathcal{V}''[v] & \text{sinon} \end{cases}$$

Définition 2.4 : la synchronisation

Soit $A = \langle D, S, F^{in}, F^{out}, dom, \Sigma, \delta, \sigma, I \rangle$ un automate de mode et $\overrightarrow{e_1}, ..., \overrightarrow{e_r}$ r vecteurs de synchronisation. Σ' est le sous-ensemble de Σ qui contient les événements présents dans les vecteurs de synchronisation. la synchronisation de A par les $\overrightarrow{e_i}$ est un automate de mode $A\ |\overrightarrow{e_1}, ..., \overrightarrow{e_r} = \langle D, S, F^{in}, F^{out}, dom, (\Sigma \setminus \Sigma') \cup \{\overrightarrow{e_1}, ..., \overrightarrow{e_r}\}, \delta', \sigma, I \rangle$ avec :

- *$\forall e \in \Sigma \setminus \Sigma'$, $s \in dom(S)$ et $I \in dom(F^{in})$ si $\delta(s, I, e)$ est définie alors $\delta'(s, I, e)$ est aussi définie et $\delta'(s, I, e) \stackrel{def}{=} \delta(s, I, e)$*
- *$\forall\ \overrightarrow{e_i} = \langle e_1, ..., e_k \rangle$, $s \in dom(S)$, et $I \in dom(F^{in})$, si $\delta(s, I, e_1), ..., \delta(s, I, e_k)$ sont définies et compatibles deux à deux, alors $\delta'(s, I, \overrightarrow{e_i}) \stackrel{def}{=} \delta(s, I, e_1) \circ_s ... \circ_s \delta(s, I, e_k)$*

Le langage AltaRica propose une notation `sync` permettant de construire des vecteurs de synchronisation. Le vecteur ainsi créé porte sur un ensemble de transitions de type :



Rappelons que les événements sont déclenchables lorsque la condition booléenne représentée par la `Garde` est vraie. La synchronisation d'événements d'automates de mode n'autorise pas le déclenchement individuel des différents événements synchronisés (il ne permet que le déclenchement simultané). Par conséquent, une synchronisation n'est déclenchable que lorsque toutes les gardes des événements synchronisés sont vraies.

L'outil Cecilia™ OCAS propose deux autres types de synchronisation : la « Diffusion » et la « DCC » (Défaillance de Cause Commune). Le type de synchronisation « Diffusion », correspond a un « *broadcast* » sur les gardes de tous les événements synchronisés et n'est déclenchable que si au moins l'une des gardes des événements synchronisés est vraie. Avec ce type de synchronisation, seuls les événements ayant leurs gardes à vrai sont déclenchables. Mais, avec la synchronisation de type « Diffusion », il n'est pas possible de tirer les événements individuellement.

Le type de synchronisation « DCC », correspond aussi à un *broadcast* sur les événements synchronisés mais, avec ce type, il est possible de tirer les événements individuellement.

Afin d'illustrer les différents types de synchronisation, prenons un exemple : deux événements `e_a` et `e_b` sont synchronisés par un nouvel événement nommé « `sync_ab` ». Considérons que pour chaque événement, il existe une garde `g` et une affectation `A`. La garde décrit les conditions à satisfaire pour tirer l'événement tandis que l'affectation représente les actions à effectuer sur les variables en cas d'occurrence de l'événement. Les deux événements sont donc définis de la manière suivante :

```
g_a |- e_a -> X_a := A_a ;
g_b |- e_b -> X_b := A_b ;
```

Pour résumer, suivant le type de synchronisation considéré, cela revient à remplacer ces deux événements par :

Synchronisation :
```
g_a and g_b |- sync_ab -> X_a := A_a, X_b := A_b ;
```
Diffusion :
```
g_a or g_b |- sync_ab ->  X_a := (if g_a then A_a),
                          X_b := (if g_b then A_b) ;
```
DCC :
```
g_a or g_b |- sync_ab ->   X_a := (if g_a then A_a),
                           X_b := (if g_b then A_b) ;
g_a |- e_a -> X_a := A_a ;
g_b |- e_b -> X_b := A_b ;
```

Nous avons ici décrit les principales caractéristiques du langage AltaRica. Nous proposerons des modèles AltaRica représentant des systèmes complexes dans les chapitres suivants.

2.2.4 Analyses et outils disponibles pour le langage AltaRica

L'évaluation de la sûreté de fonctionnement d'un système consiste à analyser les défaillances de composants pour déterminer leurs causes et estimer leurs conséquences sur le service rendu par le système.

L'analyse de sûreté de fonctionnement peut être qualitative et/ou quantitative :

L'analyse qualitative a pour but la démonstration de propriétés par l'identification de l'ensemble des scénarios menant le système d'un état de fonctionnement normal vers un état redouté. Les scénarios correspondent aux différentes combinaisons de défaillances menant à la situation redoutée.

L'analyse quantitative a pour objectif de quantifier les scénarios déterminés par l'analyse qualitative en terme de probabilité d'occurrence. C'est généralement suite à cette analyse que le système est validé ou non.

Parmi les techniques classiques les plus utilisées pour l'analyse de sûreté de fonctionnement d'un système, la simulation et la méthode d'analyse par arbres de défaillance sont les plus répandues car elles sont simples à mettre en oeuvre.

Les Arbres de Défaillances

L'analyse par Arbre de Défaillances (AdD) permet de représenter sous forme arborescente les diverses combinaisons possibles d'événements qui conduisent à la réalisation de l'événement redouté (représentant le sommet de l'arbre). Cette méthode est déductive, et la construction d'un arbre de défaillances effectuée de manière ascendante par combinaison de portes logiques et d'événements élémentaires qui mènent au sommet.

Définition 2.5 :

Un arbre de défaillance est défini comme un 1-graphe orienté ayant la propriété d'arbre (connexe et sans cycle), dont :

- *les noeuds sont soit des opérateurs logiques, soit des événements,*
- *le noeud racine est l'événement sommet noté S (événement redouté),*
- *les noeuds terminaux (feuilles) sont des événements de base notes b_i.*

La représentation graphique de l'arbre permet de représenter les niveaux successifs comme une condition logique sur les événements du niveau inférieur à l'aide d'opérateurs logiques (ET, OU, etc.). Un exemple de représentation d'arbre de défaillances est proposé en figure 2.8. L'exemple représente la formule logique $\texttt{C} \wedge (\texttt{A} \vee \texttt{B})$ menant à l'événement redouté E.

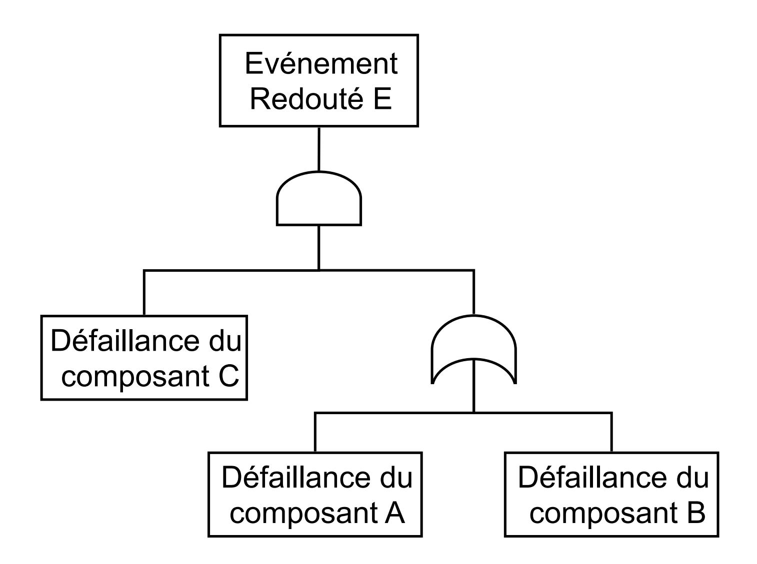


Fig. 2.8 – Exemple de représentation d'un arbre de défaillances

L'analyse qualitative par arbre de défaillances consiste à déterminer l'ensemble des coupes minimales. *Une coupe* est un sous-ensemble d'événements dont l'existence simultanée mène à l'événement redouté, et cela indépendamment de l'occurrence ou non-occurrence des autres événements de l'AdD. *Une coupe minimale* est une combinaison d'événements nécessaires et suffisants pour conduire à la situation redoutée.

La recherche des coupes minimales se fait à partir d'une transformation de l'arbre en une expression booléenne et de l'utilisation des lois de l'algèbre de Boole pour obtenir une expression booléenne réduite de l'événement redouté [RY97].

L'analyse quantitative quant-à-elle, vise à assigner, à partir des probabilités d'occurrence des événements élémentaires, la probabilité d'occurrence de l'événement redouté ainsi que des événements intermédiaires.

La méthode permettant de construire un arbre de défaillances à partir d'un modèle AltaRica est présentée dans [Rau02]. Dans cet article, Antoine Rauzy présente un algorithme permettant la compilation d'automates de mode en équations booléennes. Il présente deux intérêts pour cette compilation : un premier concernant les outils, qui sont plus efficaces sur des modèles booléens, et un second sur la génération automatique d'arbres de défaillances à partir d'une représentation de haut niveau rendant ainsi la maintenance plus aisée.

L'algorithme « naïf » possible pour décrire le principe fonctionne sur l'ensemble des états accessibles de l'automate :

1. La représentation hiérarchique du système est mise à plat en un seul automate de mode.
2. Le graphe des états accessibles est calculé.
3. Les différents chemins menant aux états redoutés sont calculés
4. Les chemins sont transformés en branches d'un arbre de défaillances (c.f. figure 2.9). Une branche est la conjonction des défaillances qui apparaissent sur un chemin. L'arbre est la disjonction des formules qui correspondent aux chemins.

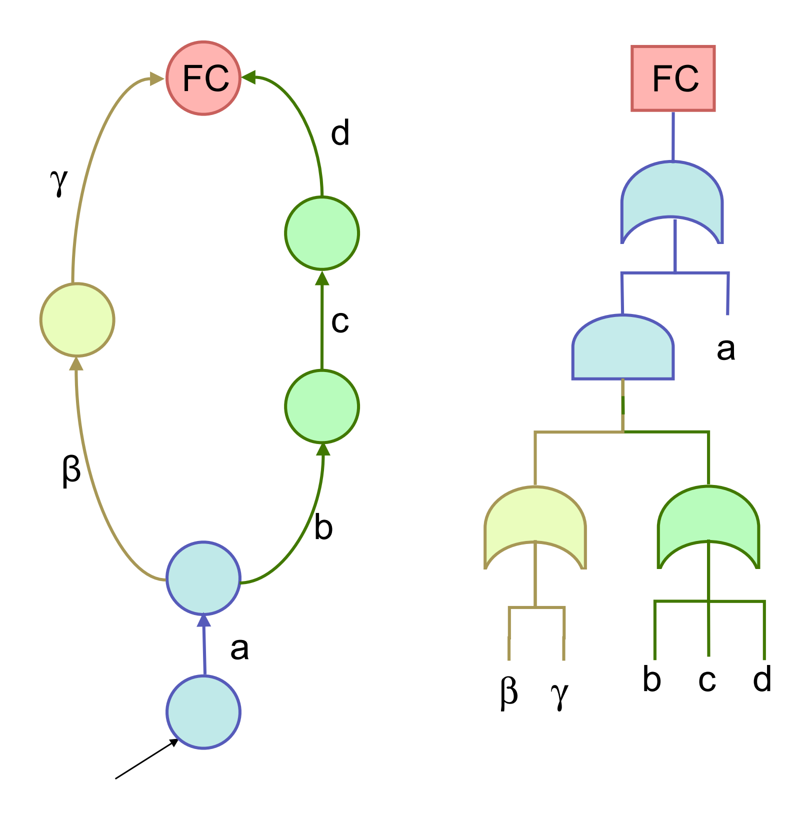


Fig. 2.9 – Transformation d'un automate de mode en un arbre de défaillances

Actuellement, il existe deux algorithmes qui optimisent l'algorithme présenté précédemment :
- celui d'Antoine Rauzy détaillé dans [Rau02],
- et celui de Philippe Thomas, auteur de l'algorithme utilisé par l'outil OCAS pour son générateur d'arbre. Mais malheureusement il n'existe pas de publication pour nous informer sur son fonctionnement.

L'analyse par arbre de défaillance est largement utilisée dans les études de sûreté de fonctionnement car elle est simple à mettre en oeuvre et elle permet d'évaluer sur le même modèle la tenue des exigences qualitatives et quantitatives.

La simulation

La *simulation* est une autre technique permettant de parcourir les différents états d'un système. Cette technique a pour principal intérêt d'exploiter directement la dynamique du système. En effet, partant de sa configuration initiale, la simulation permet de déclencher les défaillances possibles et ainsi d'observer la réaction du système permise par sa dynamique (reconfiguration, propagation, etc.).

À partir de cette technique, une génération exhaustive de l'ensemble des défaillances menant à une situation redoutée est possible. En effet, en simulant automatiquement des séquences de défaillance, il est possible d'observer si oui ou non le système atteint une situation redoutée. De plus, étant donné qu'un jeu de défaillances est défini par l'ordre d'apparition de ses défaillances, cette méthode permet d'obtenir l'ensemble des coupes menant à la situation que l'on souhaite observer. Pour la suite, on ne parlera plus de coupes, mais de *séquences*, car ces dernières conservent la propriété d'ordre d'occurrences des défaillances. De plus, pour des raisons pratiques, la génération de séquences est bornée par une longueur maximale (*ordre*) afin d'éviter l'explosion combinatoire.

La limite principale de cette méthode réside dans la génération de séquence sur des modèles de grande taille. Le temps d'obtention des différentes séquences augmente proportionnellement avec la taille du modèle et exponentiellement par rapport à la profondeur du parcours à réaliser (profondeur définie par la borne associée à la génération)[Tho06].

Les outils disponibles pour les modèles AltaRica

Les outils Cecilia OCAS et SIMFIA étant principalement dédiés à des analyses de sûreté de fonctionnement, ils intègrent des modules permettant :

- Une *simulation* « pas à pas » [KSB+04] des défaillances du système. Tout événement du système peut être tiré (individuellement ou par regroupement) et les différents impacts sont directement visibles sur le modèle. La *simulation* permet de vérifier que le système que nous modélisons possède le bon comportement.

- Un *générateur automatique d'arbre de défaillances*[Rau02] qualifié pour la certification de systèmes aéronautiques. Les arbres ainsi obtenus peuvent ensuite être utilisés par d'autres outils spécialisés pour le calcul de probabilités (Aralia, etc.) sachant que dans ce cas, nous ne nous intéressons pas à l'ordre d'apparition des défaillances.

- Un *génération de séquence* sous forme de coupes minimales. En effet, lorsque la dynamique du système n'autorise plus la génération d'arbres (le système est sujet à des reconfigurations), la génération de séquence peut prendre le relais et permet ainsi de visualiser les différentes séquences d'événements menant à une situation redoutée.

- En ajoutant des lois stochastiques aux défaillances (probabilité d'occurrence d'une défaillance pour modéliser la détérioration d'un composant au cours du temps), il est possible de faire des *simulations stochastiques* sur les modèles AltaRica et ainsi de vérifier des propriétés quantitatives sur le modèle. L'ajout de ces probabilités sur les défaillances permet aussi lors de la génération d'arbre de défaillances d'obtenir la probabilité d'occurrence d'une situation redoutée.

Notons, qu'actuellement, les outils industriels n'intègrent pas des outils de types « *Model-Checking* ».

Vers d'autres langages

Des passerelles vers d'autres langages ont également été réalisées et autorisent l'utilisation des outils de vérification et de validation disponibles pour ces langages. Une passerelle vers le langage Lustre a été réalisée au LaBRI [Gob02] et une passerelle vers le langage d'entrée du vérificateur de modèles SMV a été réalisée à l'ONÉRA-Cert [KSBC04]. En effet, après transformation du modèle AltaRica en un modèle SMV, il est ensuite possible d'utiliser les outils associés à ce langage. La technique de model-checking sur un modèle SMV [CCGR00] a été utilisée dans cette thèse.

Deuxième partie

Méthode proposée

Chapitre 3

Modélisation et Analyse de l'allocation

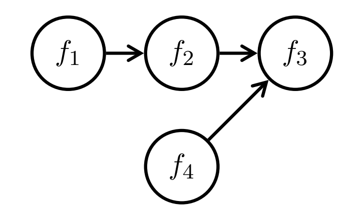
Dans cette partie, nous expliquons comment modéliser la relation d'allocation dans un modèle AltaRica puis comment vérifier les exigences sur ce modèle d'allocation. Nous justifions la correspondance faite entre la notion de *synchronisation* en AltaRica et le concept d'allocation.

Sommaire

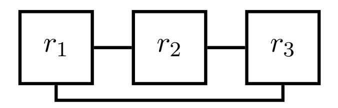
3.1 Définitions préliminaires

Le système qui nous intéresse est modélisé sous la forme d'une architecture fonctionnelle (l'ensemble des fonctions nécessaires) et d'une architecture matérielle (l'ensemble de ressources disponibles).

L'architecture fonctionnelle est caractérisée par un graphe $(\mathcal{F}, f_cnx)$ orienté pour lequel l'ensemble des noeuds $\mathcal{F} = \{f_1, ..., f_m\}$ désigne les fonctions et l'ensemble des arcs, $f_cnx \subseteq \mathcal{F} \times \mathcal{F}$, représentant les communications entre les fonctions. Un arc $r_{ij} = f_cnx(f_i, f_j)$ modélise l'envoi d'un message de f_i vers f_j.



L'architecture matérielle est définie par un graphe $(\mathcal{R}, r_cnx)$ caractérisé par l'ensemble $\mathcal{R} = \{r_1, ..., r_k, ..., r_n\}$ de n ressources connectées entre elles par les arcs $r_cnx \subseteq \mathcal{R} \times \mathcal{R}$ (on supposera dans la suite que cette relation est symétrique). Comme pour l'architecture précédente, un arc $r_{ij} = r_cnx(r_i, r_j)$ représente l'échange de données entre les ressources r_i et r_j. Cette architecture matérielle permet de représenter une architecture de différentes ressources de calcul communicantes entre elles par un réseau.



La relation d'allocation est représentée par *alloc* qui, pour une fonction f de $\mathcal{F}$ et une ressource r de $\mathcal{R}$, vaut 1 si et seulement si f est allouée sur r.

$$\begin{aligned} alloc: \quad & \mathcal{F} \times \mathcal{R} \rightarrow \{0,1\} \\ & alloc(f, r) = 1 \end{aligned}$$

Exemple : Une relation d'allocation possible de l'architecture fonctionnelle sur l'architecture matérielle de la figure précédente est donnée dans la figure suivante :

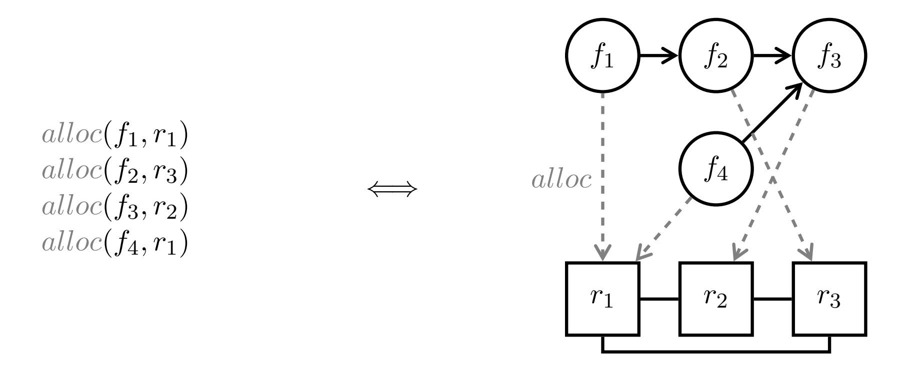


Fig. 3.1 – Exemple d'une relation d'allocation

Cette allocation permet de représenter la répartition des différentes fonctions sur les ressources.

Lorsque nous étudions séparément le modèle d'architecture fonctionnelle du modèle d'architecture matérielle, nous supposons que chaque fonction est implicitement allouée sur une ressource unique, totalement indépendante (n'ayant aucune interaction avec les autres ressources). Ce type de modélisation de la relation d'allocation permet d'analyser et de vérifier des propriétés du modèle d'architecture fonctionnelle sans se soucier de l'architecture matérielle utilisée pour supporter le système.

Dans la suite de ce chapitre, nous utilisons un exemple très simple baptisé *COM/MON* pour illustrer les notions présentées. La figure 3.2 propose une représentation du modèle *COM/MON*.

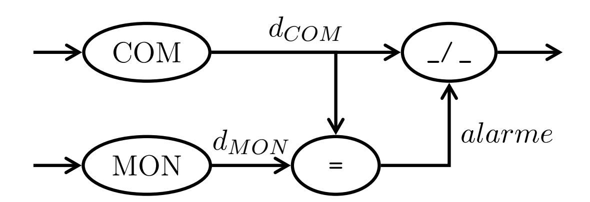


Fig. 3.2 – Modèle *COM/MON*

Ce modèle est composé de 2 fonctions séparées, l'une appelée *COM* (Commande) et l'autre *MON* (Monitoring), qui effectuent un calcul identique à partir de leurs entrées. Une comparaison est effectuée à partir des résultats fournis par ces fonctions. Dans le cas où les fonctions ne produisent pas des valeurs identiques, un mécanisme doit interrompre le transfert des sorties de la fonction *COM* vers l'extérieur.

3.2 Modélisation de l'architecture fonctionnelle

La représentation de l'architecture fonctionnelle est, comme nous l'avons définie précédemment, faite à partir d'un graphe orienté représentant les différentes fonctions du système ainsi que leurs interactions (interconnexions). La modélisation en AltaRica consiste à traduire chaque noeud de $\mathcal{F}$ en un composant AltaRica et chaque arc qui représente une communication entre deux noeuds en connexion de variables de flux entre deux composants.

L'intérêt de l'utilisation du langage AltaRica réside dans la possibilité d'enrichir la description des différents composants pour se rapprocher du comportement attendu des fonctions sur une architecture matérielle.

Voyons maintenant comment appliquer ces règles pour modéliser l'exemple du *COM/MON*. Comme présenté dans la figure 3.2, ce système est composé de quatre composants. Parmi ces quatre composants, deux sont identiques du point de vue de leur comportement car ils effectuent tout deux une tâche de calcul identique à partir de leur entrée. Ce composant de calcul possède donc une entrée et une sortie pour faire transiter une donnée après un calcul sur cette dernière. Comme pour la plupart des fonctions de calcul, nous souhaitons lui associer un certain comportement face aux défaillances. Ce comportement peut être schématisé par l'automate suivant :

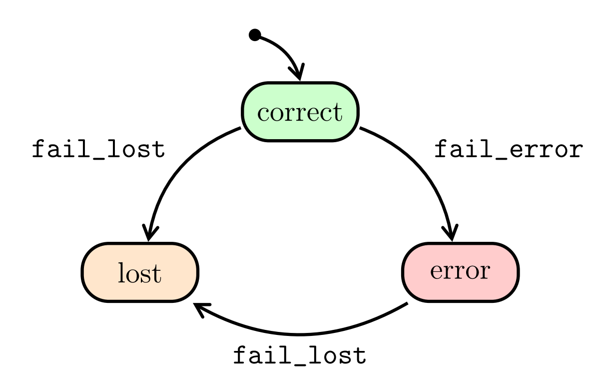


Fig. 3.3 – Comportements possibles des fonctions

Cet automate décrit le comportement des fonctions en présence de défaillances. L'apparition d'une défaillance (`fail_error` ou `fail_lost`) amène un changement d'état de la fonction impactée. En effet, la fonction impactée n'est plus dans son état nominal qui correspond à son bon fonctionnement mais dans un état décrivant le dysfonctionnement. Ainsi, les deux autres états de cet automate décrivent les deux états de dysfonctionnement :

1. L'état **error** représente un dysfonctionnement de la fonction entraînant la production d'une donnée incorrecte.

2. L'état **lost** représente un dysfonctionnement qui mène à l'absence de donnée. Lorsque la fonction est dans cet état, elle n'est plus capable de fournir de données.

De la même manière, une représentation particulière du type de données doit être imposée dans la modélisation. En effet suivant les choix de modélisation, la représentation des échanges d'informations peut différer. En AltaRica, ces échanges d'informations sont modélisés par des flux de données permettant une visualisation de la propagation de ces données dans le système. Par exemple, dans un système réel, généralement les capteurs fournissent une donnée en fonction de leur mode de fonctionnement. En effet, un capteur fonctionnant correctement fournit en permanence une donnée correcte. Si ce dernier tombe en panne, soit il ne fournit plus aucune donnée, soit il fournit des données erronées. Nous avons choisi de ne pas représenter la valeur réellement produite par la fonction, mais plutôt d'étudier une valeur abstraite qui dénote l'occurrence d'une défaillance. À partir de cette constatation, nous avons modélisé en AltaRica cette valeur abstraite à l'aide d'un type énuméré pouvant prendre 3 valeurs distinctes.

1. `correct` : la donnée n'a pas subi de défaillance
2. `erroneous` : la valeur de la donnée est erronée
3. `lost` : la valeur de la donnée n'est pas produite

Les évolutions possibles de ces comportements peuvent être modélisées en AltaRica de la façon suivante :

```
node fonct
 state s:{correct,error,lost};
 flow
  O1:out:{correct,error,lost};
  I1:in:{correct,error,lost};
 event
  fail_error,
  fail_lost;
 trans
  not(s = lost) |- fail_lost -> s:=lost;
  s = correct |- fail_error -> s:=error;
 assert
  O1 = case {
    s = correct : I1,
    else s
   };
 init s:=correct;
edon
```

Nous obtenons de la même manière le code AltaRica des autres fonctions. La traduction du système complet passe par la création d'un noeud `Main` qui permet d'interconnecter les différentes fonctions à l'aide de relations entre les variables de flux des composants AltaRica correspondants.

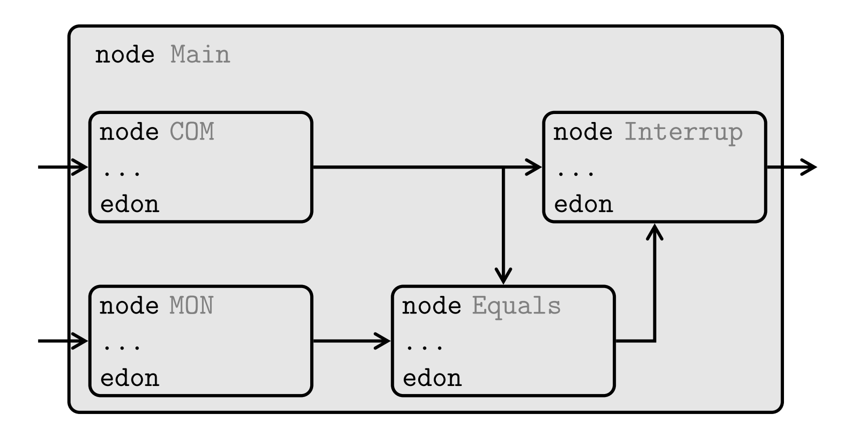


Fig. 3.4 – Représentation du noeud `Main` du *COM/MON*

Les connexions entre les différents noeuds du graphe fonctionnel doivent être reportées sur le modèle AltaRica. Ainsi, chaque arc du graphe doit être représenté par une liaison entre modèles AltaRica. De plus, le graphe fonctionnel étant orienté, les liaisons créées doivent de même orienter leur flux. Finalement, pour tout arc appartenant à l'ensemble f_cnx des arcs entre les fonctions, où f_i et f_j sont deux fonctions connectées, tel que l'orientation est de f_i vers f_j, la traduction en AltaRica doit préserver la correspondance entre les flux entrants dans le composant f_j et les flux sortants du composant f_i.

$$\forall (f_i, f_j) \in f_cnx, f_i \rightarrow f_j : F_{f_j}^{in} = F_{f_i}^{out}$$

En appliquant cette dernière règle de transformation à toutes les fonctions du système *COM/MON*, nous obtenons le code AltaRica correspondant au noeud global `Main` suivant :

```
node Main
 sub
  COM : fonct,
  MON : fonct,
  Equals : equals,
  Interrup : interrup;
 assert
  Interrup.in_com = COM.out;
  Interrup.in_equals = Equals.out;
  Equals.in_com = COM.out;
  Equals.in_mom = MON.out;
edon
```

Lors de la lecture du code AltaRica du noeud Main, nous remarquons que les composants `Equals` et `Interrup` ne sont pas déclarés comme « `fonct` ». Ce choix est volontaire, car ces 2 composants n'ont pas le même comportement qu'une fonction de calcul lorsqu'ils sont en présence de défaillances.

1. Le composant `Equals` qui effectue la comparaison des données fournies par la fonction *COM* et la fonction *MON*, n'est pas sujet aux défaillances. Nous supposons que cette comparaison s'effectue toujours sans la présence d'une défaillance. Le comportement de ce composant est donc limité au signalement par un booléen de la différence de valeurs de ses deux entrées. Le comportement de ce composant possède une limitation en cas de présence de deux valeurs erronées similaires. En effet, dans le cas de deux données erronées identiques, ce composant ne peut pas faire la différence avec un cas de deux données correctes et identiques. Dans ce cas, il considère bien que les valeurs sont identiques, mais il est incapable d'évaluer le côté erroné des données.

   Le code AltaRica implémentant un tel composant est le suivant :

```
node equals
 flow
   O1:out:bool;
   I1:in:{correct,error,lost};
   I2:in:{correct,error,lost};
 assert
   O1 = case {
     I1 = I2 : true,
    else false
   };
edon
```

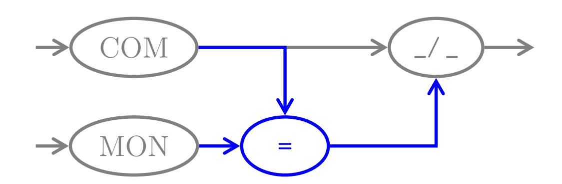


Fig. 3.5 – Le composant `Equals`

2. Le composant `Interrup` quant à lui, permet comme son nom l'indique d'interrompre le flux de données fournit par *COM*. Cette interruption est activée par le signal (la variable booléenne) provenant de `Equals` lorsque les valeurs des données produisent par COM et MON sont différentes. Le code AltaRica mettant en œuvre ce composant est le suivant :

```
node interrup
 flow
   O1:out:{correct,error,lost};
   I1:in:{correct,error,lost};
   I2:in:bool;
 assert
   O1 = case {
     I2 = false : lost,
    else I1
   };
edon
```

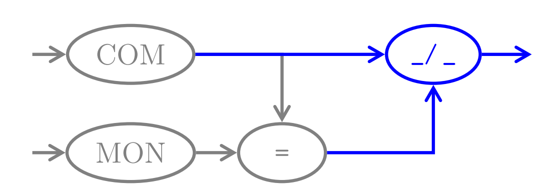


Fig. 3.6 – Le composant `Interrup`

Voyons maintenant un exemple d'analyse vérifiant que le modèle AltaRica global du système possède bien le comportement souhaité.

Afin de tester notre modèle, il faut le compléter par deux sources (s_1,s_2) qui vont fournir aux fonctions *COM* et *MON* les données nécessaires. Nous allons, dans cet exemple, vérifier que le couple de composants `Equals`, `Interrup` permet lors d'une défaillance sur la fonction *COM* ou *MON* de ne pas propager une donnée erronée. Pour vérifier cette propriété, un observateur `Obs` est ajouté en sortie de Interrup pour contrôler la donnée fournie par ce composant. Le modèle d'exemple à tester devient alors le suivant :

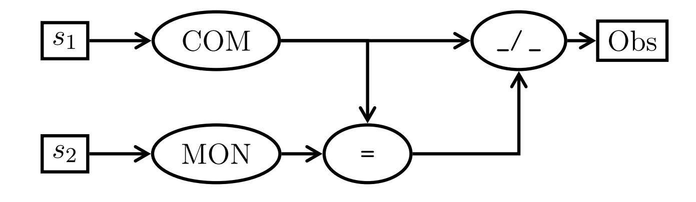


Fig. 3.7 – Exemple utilisé pour l'analyse du COM/MON

Lors de la première exécution de notre modèle, les différentes valeurs en sortie de nos composants sont reportées dans le tableau suivant :

sources	fonctions	Equals	Obs
s_1 : `correct` s_2 : `correct`	COM : `correct` MON : `correct`	true	`correct`

L'analyse des valeurs du tableau montre que lorsque le système fonctionne, la donnée qui est

observée par le composant `Obs` est bien correcte. Voyons maintenant la réaction du système face à une défaillance (`fail_error`) sur le composant *COM*.

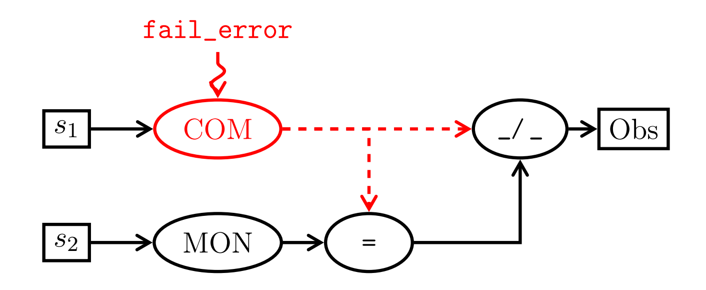


Fig. 3.8 – Injection d'une défaillance sur *COM*

Cette défaillance a comme effet le changement de valeur de la donnée calculée par *COM*. Cette valeur modifiée va avoir une influence sur les composants qui manipulent cette donnée. Regardons les valeurs observées de ce système.

sources	fonctions	Equals	Obs
s_1 : `correct` s_2 : `correct`	COM : `erroneous` MON : `correct`	false	`lost`

L'analyse des valeurs observées démontre bien que notre système possède le comportement attendu. En effet, nous remarquons que le composant `Equals` signale la différence des données (la valeur de sa variable de sortie vaut `false`) et le composant `Interrup` interrompt bien le transfert de la donnée provenant de *COM* puisque la valeur observée par *Obs* est `lost` (valeur qui dénote l'absence de donnée).

Une limite possible à ce comportement est atteinte lorsque des données erronées arrivent sur les 2 entrées du composant `Equals`, il n'est plus capable de détecter que la valeur est erronée. On suppose que les valeurs sont erronées de façon cohérente, c.-à.-d. qu'elles ont la même valeur. Le comportement de ce composant est donc limité à identifier une différence parmi ses entrées, peu importe leurs valeurs.

sources	fonctions	Equals	Obs
s_1 : `erroneous` s_2 : `erroneous`	COM : `erroneous` MON : `erroneous`	true	`erroneous`

Par conséquent, la limite du *COM*/*MON* est qu'il ne tolère pas le double erroné . En effet comme vu dans le tableau précédent, ce mécanisme ne peut empêcher la propagation d'une donnée erronée lorsque les deux sources (*COM* et *MON*) fournissent une donnée erronée.

Une autre limite semblable du comportement de ce composant est visible en présence des défaillances `fail_lost`.

sources	fonctions	Equals	Obs
s_1 : `lost` s_2 : `lost`	COM : `lost` MON : `lost`	true	`lost`

En effet, en cas de cette défaillance sur les sources ou les fonctions COM et MON, le composant n'est pas capable de détecter l'absence de donnée engendrée par ces défaillances et par conséquent, la propagation ne sera pas stoppée.

3.3 Modélisation de l'architecture matérielle

3.3.1 Architecture matérielle informatique

La modélisation de l'architecture matérielle s'effectue de la même façon que l'architecture fonctionnelle, mis à part le fait qu'il n'y a que deux types de ressources : ressource de communication (« *Bus* » représentant la relation r_cnx de l'architecture matérielle) et ressource de calcul (« *Cpu* » représentant un élément de $\mathcal{R}$).



(a) Ressource de communication (b) Ressource de calcul

FIG. 3.9 – Ressources matérielles

Une ressource de communication (fig. 3.9(a)) permet le transport des différentes données d'un système. Les défaillances que nous considérons comme intéressantes pour cette ressource sont :

- La corruption des données transportées due à une erreur dans le protocole de communication (par exemple, une interférence électromagnétique peut entraîner un dysfonctionnement dans la transmission, et par conséquent les données transmises seraient erronées). Cette défaillance correspond dans la réalité à une erreur sur quelques bits des données lors d'une transmission. De plus, cette défaillance n'est pas nécessairement permanente.
- Une perte complète de la ressource due à une défaillance comme par exemple, une coupure du câble nécessaire à la transmission entraîne l'interruption totale de la communication.

Une ressource de calcul (fig. 3.9(b)) permet d'exécuter les différentes tâches du système. Tout comme la ressource de communication, cette ressource peut subir des défaillances. Nous considérons deux défaillances possibles :

- Une défaillance pour exprimer une erreur intempestive du calculateur, c.-à-d. que durant un laps de temps très court, le calculateur va produire une donnée erronée. Ces défaillances peuvent apparaître lorsque les calculateurs sont amenés à exécuter plusieurs tâches en parallèle.
- Une défaillance permanente permettant de modélisation la perte totale de la ressource. La perte totale d'un calculateur correspond par exemple à la perte de l'alimentation nécessaire à la ressource. Il est certain que si la ressource n'est plus alimentée, alors elle est dans l'incapacité d'exécuter une tâche.

Ainsi, chaque composant représentant une ressource matérielle présente deux types de défaillances : `fail_error` et `fail_lost`. Ces défaillances permettent de modéliser les pannes qui nous intéressent. À savoir : la défaillance `fail_lost` qui permet de représenter la perte totale de la ressource (perte de l'alimentation électrique, détérioration interne de la ressource, etc.). Une fois que cette défaillance apparaît, la ressource n'est plus utilisable (elle ne communique plus et par conséquent ne fournit plus de données). La défaillance `fail_error` quant à elle, représente une perte « partielle » de la ressource. Lorsqu'une ressource subit cette défaillance, toutes les données qui sont utilisées ou manipulées par cette ressource seront considérées comme corrompues (erronées). L'événement `update` permet de revenir à un état de fonctionnement correct.

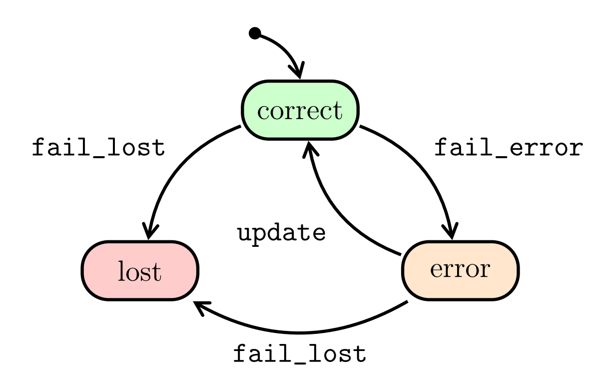


Fig. 3.10 – Comportements possibles des ressources

Le code AltaRica correspondant à une ressource matérielle est le suivant :

```
node ress
 state
  S:{correct,error,lost};
 event
  fail_error, fail_lost, update;
 trans
  not(S = lost) |- fail_lost -> S:=lost;
  S = correct |- fail_error -> S:=error;
  S = error |- update -> S:=correct;
 init
  S:=true;
edon
```

Ce premier modèle nous donne le comportement souhaité face aux défaillances, mais il ne donne pas le comportement dû aux interconnexions.

Les connexions entre les ressources ne sont pas représentées dans le code AltaRica car elles varient en fonction de l'architecture étudiée. En effet, suivant les choix d'architecture, certains calculateurs ne peuvent échanger des données qu'avec certaines ressources de communication (l'utilisation des protocoles varie suivant les débits d'informations et le type d'information qui circule).

Ainsi, les connexions des ressources de communication dépendent du nombre de ressources de calcul connectées. Lors de la réalisation de l'architecture, il faut modifier le code AltaRica de chaque ressource de communication afin d'y ajouter les bonnes connexions entrantes et sortantes des autres ressources connectées.

Prenons maintenant l'exemple d'une architecture permettant de supporter le système COM/MON de la section précédente. Elle est composée de 2 ressources de communication (Bus_1, Bus_2) et 4 ressources de calcul (R_1, R_2, R_3, R_4). Considérons que R_1 produise certaines quantités de données ne pouvant être véhiculées que par une ressource de communication de haut débit, considérons aussi que les 2 calculateurs R_3 et R_4 doivent s'échanger des données afin de se partager le calcul d'une tâche importante et considérons enfin que la ressource R_2 permet de faire la liaison entre les données produites par R_1 et celles produites par le couple R_2, R_3.

Une telle architecture peut être représentée par la figure suivante :

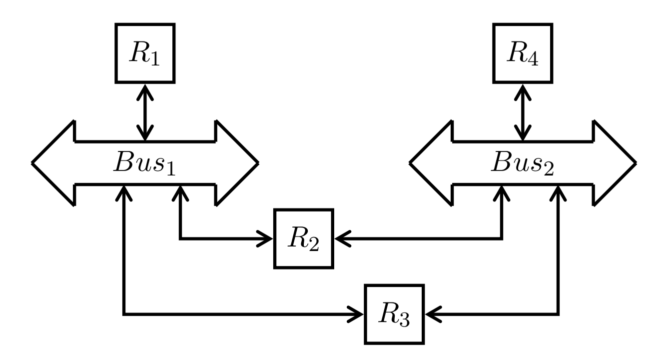


Fig. 3.11 – Exemple d'architecture matérielle

Comme vu dans la figure, les connections entres les différentes ressources sont considérées comme « bi-directionnelles », c.-à.-d. qu'une ressource ne se contente pas d'écrire sur la ressource mise à sa disposition pour communiquer, elle doit aussi pouvoir lire des données circulant sur cette dernière.

L'analyse de cette architecture permet d'identifier certains regroupements de ressources ayant le même comportement et par conséquent le même code AltaRica. Ainsi, R_1 et R_4 peuvent être regroupées car elles possèdent chacune une seule connexion vers une ressource de communication. R_2, R_3 peuvent elles aussi être regroupées car elles sont toutes les deux connectées à deux autres ressources. Les ressources de communications Bus_1 et Bus_2 quant à elles, peuvent être regroupées car elles possèdent trois connexions avec d'autres ressources.

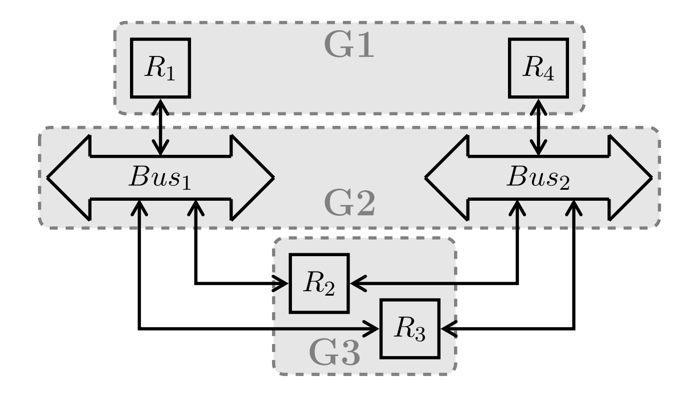


Fig. 3.12 – Regroupements possibles de composants

Ainsi, seulement trois noeuds AltaRica suffisent pour modéliser cet exemple. Partant du comportement « générique » défini précédemment, on peut en déduire le code de chaque groupe.

Le premier groupe (**G1**) est composé de deux ressources de calcul identiques ne possédant qu'une entrée/sortie vers une autre ressource. Il suffit donc de lui ajouter le comportement souhaité de ses sorties en fonction de l'état interne du composant. Toujours dans l'idée d'associer à chaque composant un comportement générique, il faut spécifier que chaque composant (qu'il soit un composant de calcul ou de communication) possède le comportement d'une ressource (`ress`) défini précédemment. Cette spécification est possible en utilisant la notion de *hiérarchie* en AltaRica. Elle permet d'inclure dans un composant un composant déjà défini. Dans la modélisation de l'architecture matérielle, nous souhaitons que la ressource informe les autres ressources connectées de son état interne. Ce comportement est possible en considérant que la sortie du composant n'est utilisée que pour renvoyer l'état de la ressource incluse (`O1` renvoie la valeur de `ress.S`)

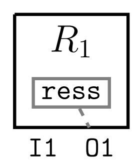


Fig. 3.13 – Exemple de représentation d'une ressource

La connexion de R_1 vers Bus_1 est représentée dans la figure 3.14 par une flèche bidirectionnelle. Cette flèche permet d'exprimer graphiquement que la connexion s'effectue dans les deux sens : de R_1 vers Bus_1, mais aussi de Bus_1 vers R_1. Lors de la modélisation en AltaRica, il faut donc ajouter pour chaque liaison entre les composants un port représentant une connexion entrante et un port représentant une connexion sortante.

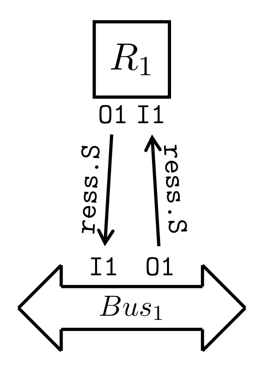


Fig. 3.14 – Exemple de connexion entre les ressources

Le code AltaRica correspondant aux ressources du groupe G1 est le suivant :

```
node CPU1
 flow
   I1:in:{correct,error,lost};
   O1:out:{correct,error,lost};
 sub
   r:ress;
 assert
   O1 = case { r.s=correct : I1,
               else r.s } ;
edon
```

Fig. 3.15 – Les ressources de calcul « simples »

Les composants AltaRica des 2 autres groupes (**G2**, **G3**) sont modélisés de la même façon, ils ont un code semblable mis à part leur nombre de connexions entrantes.

Pour tout composant ayant n connexions, on construit le noeud AltaRica correspondant. Pour cela, il suffit, d'ajouter au composant « générique » les entrées et d'exprimer la valeur de sortie en fonction de l'état interne de `ress` et des valeurs de ses entrées. En effet, le comportement du composant que nous souhaitons appliquer et qu'en cas d'une donnée erronée sur l'une de ses entrées, cette dernière doit être propagée en sortie et nous supposons également que ce composant doit obtenir une valeur de toutes ses entrées pour produire une donnée.

$\forall i \in [1, n]$:
- I_i `: in : {correct,error,lost}` ;
- O `: out :{correct,error,lost}` ;
- O `= case { r.S = correct and (I1=erroneous or ... or In = erroneous ) : erroneous,`

```
r.S = correct and (I1=lost or ... or In=lost : lost,
r.S = correct and I1=correct and ... and In=correct : correct,
else r.S };
```

3.3.2 Architecture spatiale

Il est aussi possible d'appliquer notre principe de modélisation et de vérification sur d'autres types de systèmes. Par exemple en ce qui concerne le problème d'allocation des différents équipements dans un avion, il est possible de faire une abstraction des différentes zones d'un avion pour n'obtenir qu'un graphe décrivant l'agencement des zones.

Considérons qu'une zone d'un avion soit représentée par un cube. Les différentes faces de ce cube représentent les différentes connexions possibles avec les autres zones. Ainsi comme illustré dans la figure 3.16, les noms des faces expriment les positions possibles des zones adjacentes. Partant de cette considération, une mise à plat de ce cube est possible en prenant quatre faces comme les quatre côtés du carré avec en plus deux connexions au niveau de deux coins.

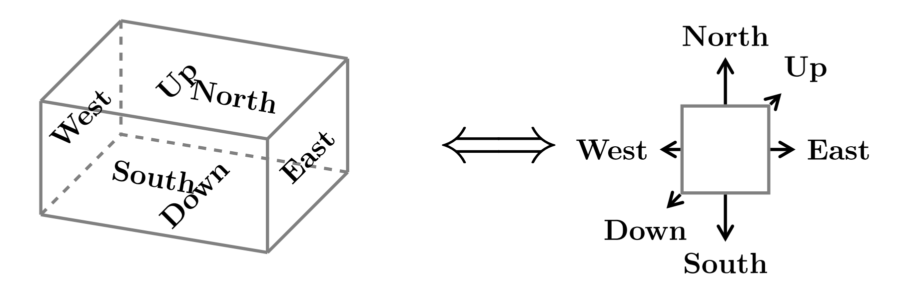


Fig. 3.16 – Composant représentant une zone avion

Cette modélisation par un graphe des zones d'un avion permet d'appliquer notre technique sur le problème de l'allocation spatiale. En effet, partant de ce graphe, il est possible de le considérer comme une architecture matérielle. Ce graphe permet de résoudre le problème de placement des équipements dans un avion et donc considérer le problème comme un problème d'allocation d'équipements sur des zones de l'avion.

Le comportement AltaRica représente les défaillances associées à une zone de l'avion comme l'impact d'un débris de pneu ou de réacteur qui conduisent à la perte des équipements placés dans cette zone, ou comme l'agression électromagnétique qui va corrompre les données transmises (ou stockées) par les équipements positionnés dans cette zone. Ce comportement permet de retrouver les défaillances et leurs effets du composant `ress` présenté précédemment. En effet, la défaillance correspondant à la perte totale des équipements présents dans la zone représente bien l'influence de la défaillance `fail_lost` et la défaillance qui influence le dysfonctionnement des équipements correspond bien aux effets provoqués par `fail_error`.

Le composant AltaRica correspondant possède autant de flux de sortie qu'il a de connexions avec les autres zones (par exemple, un port `O_N` pour représenter sa connexion avec la zone du nord, un port `O_W` pour repésenter sa connexion avec la zone située à l'ouest...) et il renvoie sur ses sorties l'information d'un éventuel impact. Une représentation graphique pour un tel composant est donnée par la figure 3.17.

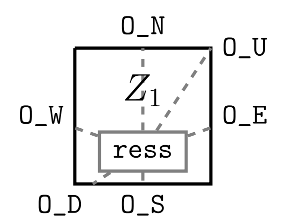


Fig. 3.17 – Exemple de représentation d'une zone

Le code AltaRica d'un tel composant peut s'écrire ainsi :

```
node Zone
 flow
   O_N,O_S,O_E,O_W,O_U,O_D: out:{correct,error,lost};
 sub
   z:ress;
 assert
   O_N:= z.S;
   O_S:= z.S;
   O_E:= z.S;
   O_W:= z.S;
   O_U:= z.S;
   O_D:= z.S;
edon
```

```
node ress
 state
  S:{correct,error,lost};
 event
  fail_error, fail_lost, update;
 trans
  not(S = lost) |- fail_lost -> S:=lost;
  S = correct |- fail_error -> S:=error;
  S = error |- update -> S:=correct;
 init
  S:=correct;
 edon
```

Ces composants modélisant les différentes zones d'un avion ainsi que les connexions permettant de les relier, facilitent la représentation d'un avion sous la forme d'un graphe de zones adjacentes. Un exemple de graphe est donné par la figure suivante :

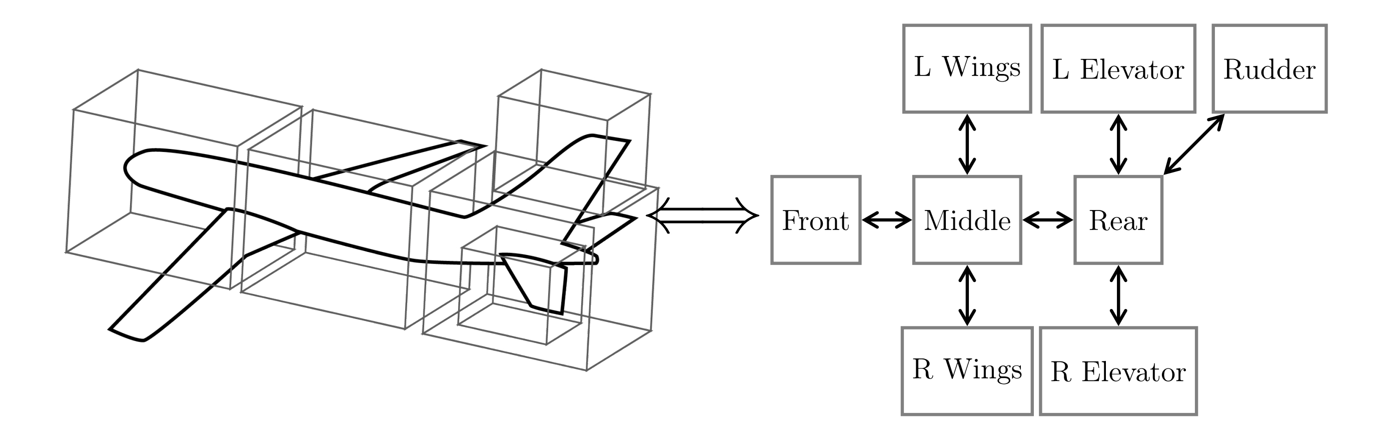


Fig. 3.18 – Représentation possible des zones d'un avion

3.4 Modélisation de l'allocation

Lorsque nous modélisons en AltaRica la relation d'allocation, nous souhaitons observer le comportement de l'architecture fonctionnelle en présence de défaillances matérielles. Nous construisons tout d'abord un modèle global basé sur les défaillances des composants de l'architecture fonctionnelle et des composants de l'architecture matérielle. Ces défaillances n'ont à ce stade de la modélisation aucune liaison. La relation d'allocation permet donc de relier ces différentes défaillances pour illustrer l'influence des architectures dans un modèle commun. En effet, les seules défaillances observables étaient des défaillances élémentaires sur les fonctions qui composent notre système. En ajoutant la relation d'allocation à notre modèle, nous construisons un modèle global enrichi par des relations entre les défaillances des modèles des architectures fonctionnelles et matérielles.

La relation d'allocation permet de représenter l'association de différentes fonctions sur une même ressource. Cette relation doit illustrer que lorsqu'une défaillance survient sur une ressource, alors toutes les fonctions allouées doivent aussi subir cette même défaillance. Nous pouvons ainsi observer la réaction de notre système face à la défaillance d'une ressource permettant l'exécution du système.

Cette relation introduit une nouvelle notion : la « défaillance de cause commune » (ou de mode commun). Lorsque plusieurs composants dépendent d'éléments communs (par exemple, plusieurs composants peuvent dépendre d'une même source électrique), et qu'une défaillance intervient sur un de ces éléments, tous les éléments communs vont subir cette même défaillance. Par exemple si nous prenons l'exemple d'éléments connectés à une même source électrique, une défaillance qui entraîne la perte de cette source est aussi une défaillance qui entraîne un dysfonctionnement des éléments connectés (sous réserve de considérer ces éléments comme des entités indépendantes lors de la modélisation).

Nous utilisons la synchronisation d'événements présentée dans la section 2.2.3 afin de modéliser l'allocation. La synchronisation relie les défaillances de l'architectures matérielle avec les défaillances de l'architecture fonctionnelle. Cet opérateur du langage AltaRica ajoute un événement supplémentaire qui permet de simplifier l'exécution de tous les événements synchronisés. En effet, ce nouvel événement effectue un « broadcast »[1] sur tous les événements présents dans la synchronisation, c.-à.-d. que ce nouvel événement va se substituer à chaque événement présent dans la synchronisation et lorsque la garde le permet, il va s'appliquer.

Voyons à présent le comportement d'une telle synchronisation. Pour cela, prenons l'exemple de trois tâches (`Fxl_1`, `Fxl_2`, `Fxl_3`) allouées sur une même ressource (`Ress`). Le comportement souhaité d'une telle allocation est que la perte de la ressource engendre la perte des trois tâches.

Pour cela une première synchronisation introduit un nouvel événement de type « broadcast » : *Fxl_fail* permettant de représenter la perte simultanée des tâches concernées. En effet, cet événement s'applique à toutes les tâches allouées, mais ne être doit tiré que lorsque ces dernières l'autorisent (lorsque les gardes permettent cet événement). Sachant que les différentes tâches allouées ne sont pas toujours actives, cette « forme » de synchronisation permet d'appliquer les défaillances seulement aux tâches actives.

Une seconde synchronisation doit être ajoutée pour, cette fois-ci, simuler la perte de la ressource ainsi que l'ensemble de ses tâches qui y sont allouées : cette nouvelle défaillance est nommée *alloc_fail*. Contrairement à la synchronisation précédente, celle-ci doit impérativement changer l'état de la ressource lorsqu'elle apparaît et de plus, elle doit déclencher la première synchronisation pour impacter les fonctions actives allouées à cette ressource. Ainsi, cette nouvelle défaillance (introduite par cette nouvelle synchronisation) ne doit être possible que lorsque la garde de la ressource l'autorise.

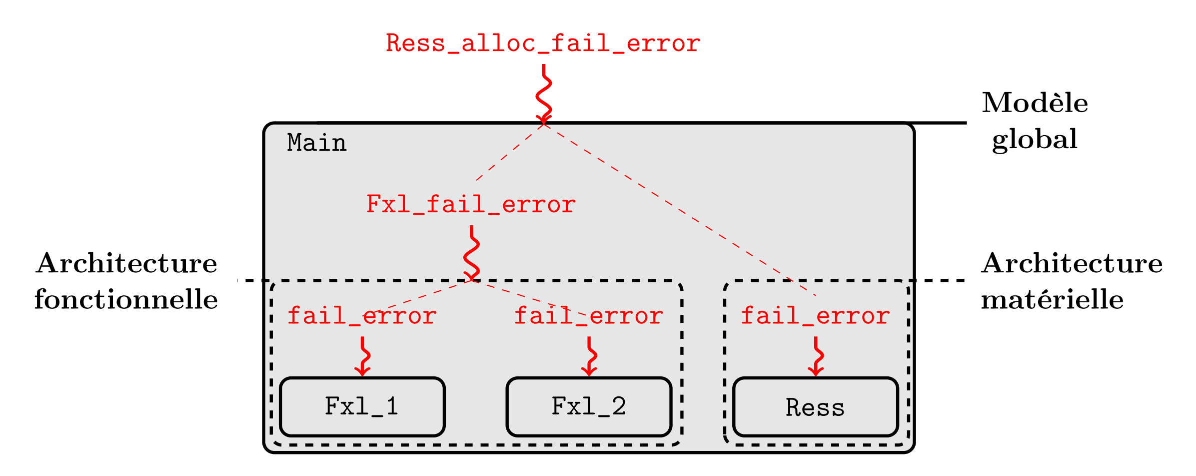
Voyons à présent comment exprimer en AltaRica un tel comportement face aux défaillances. Considérons que chaque composant possède une défaillance (`fail_error`). Nous souhaitons ajouter à notre

[1] le *Broadcasting* désigne une méthode de diffusion de données à partir d'une source unique vers un ensemble de récepteurs

modèle une défaillance supplémentaire qui représente la défaillance simultanée des deux tâches lors de la défaillance de la ressource.

Comme vu précédemment, il faut ajouter dans un premier temps une défaillance (synchronisation) `Fxl_fail_error` afin d'exprimer le comportement erroné simultanément de `Fxl_1` et `Fxl_2`, et dans un second temps il faut synchroniser cette dernière avec la défaillance de `Ress`. Cette nouvelle défaillance correspond à un nouvel événement nommé `Ress_alloc_fail_error`. Pour permettre la synchronisation de tous ces événements, il faut définir un élément intermédiaire permettant de rassembler les fonctions en un même composant (`a_Fxl`). Puis il faut ajouter à cet élément une nouvelle défaillance permettant de synchroniser la perte simultanée des fonctions (`Fxl_fail_error`). Cet élément regroupe donc l'architecture fonctionnelle dans un même composant (un élément de ce type est appelé *équipement*).

Une représentation possible de la modélisation souhaitée est donnée par la figure suivante :



Le code AltaRica du système ainsi que des composants est le suivant :

```
node Main
 event
  Ress_alloc_fail_error;
 sub
  a_Fxl : Archi_Fxl,
  Ress : ress,
 sync
  <Ress_alloc_fail_error, a_Fxl.Fxl_fail_error, Ress.fail_error>,
edon
```

```
node Archi_Fxl
  event
    Fxl_fail_error;
  sub
    FXL_1 : fonct;
    FXL_2 : fonct;
  sync
    <Fxl_fail_error : Fxl_1.fail_loss or Fxl_2.fail_loss>,
    <Fxl_1.fail_loss>,
    <Fxl_2.fail_loss>;
edon
```

```
node fonct
 ...
edon
```

```
node ress
 ...
edon
```

En ce qui concerne les défaillances `fail_lost`, la technique est exactement la même.

3.4.1 Application au COM/MON

Voyons à présent comment vérifier une allocation proposée pour l'architecture fonctionnelle de l'exemple du COM/MON (fig 5.1).

L'allocation que nous allons vérifier suit les directives suivantes :
- Comme les composants de la voie COM (S_1 et COM) possèdent une forte dépendance, ils sont allouées sur une même ressource.
- Les composants de la voie MON (S_2 et MON) sont eux aussi regroupés.
- Les composants `Interrup` et `Equals` doivent pouvoir récupérer des données provenant des deux voies distinctes.

Une première architecture à vérifier est constituée de trois ressources de calcul, toutes connectées à un même bus.

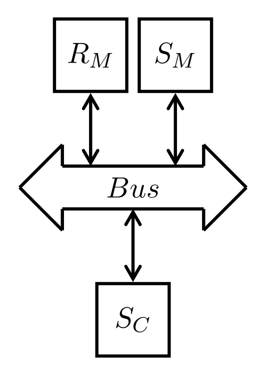


Fig. 3.19 – Première Architecture pour le COM/MON

L'allocation que nous souhaitons vérifier doit respecter plusieurs directives. Celles que nous avons choisies sont les suivantes :
- allocation du groupe correspondant à la voie COM (S_1 et COM) sur la ressource S_C,
- allocation de la fonction `Equals` sur la ressource R_C
- allocation du groupe correspondant à la voie MON (S_2 et MON) sur la ressource S_M,
- allocation de la fonction `Interrup` sur la ressource R_M.

Cette allocation peut être illustrée par la figure suivante :

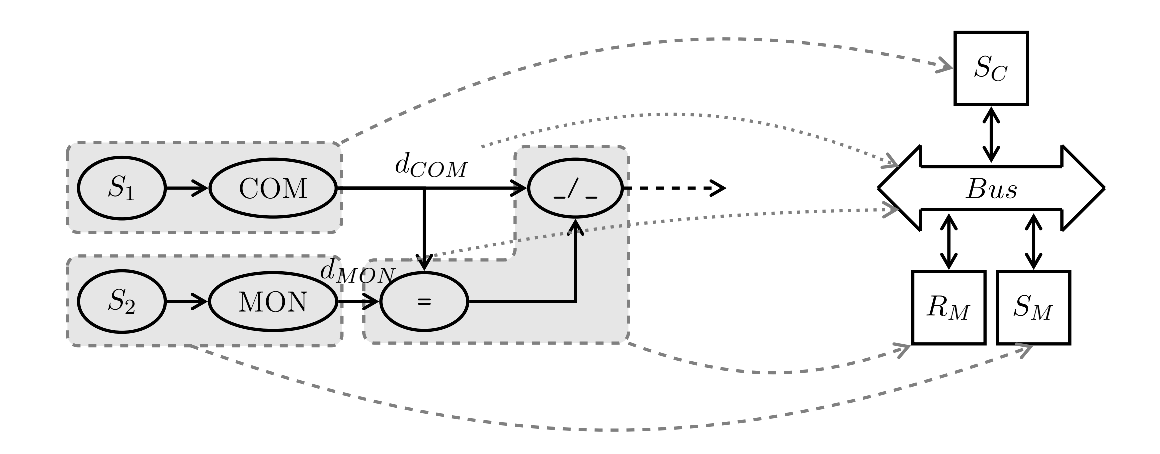


Fig. 3.20 – Une allocation possible pour le COM/MON

Cette architecture n'est pas valable, car il est facilement compréhensible que faire transiter toutes les données par une même ressource de type *Bus* n'est pas une bonne solution. En effet, le comportement erroné de ce centralisateur de données suffit à rendre inutiles les mécanismes fonctionnels mis en place par l'architecture fonctionnelle. Ce comportement erroné est atteint en présence de la défaillance `Bus_alloc_fail_error`.

```
node Main
 evt
  Bus_alloc_fail_error,...;
 sync
  <Bus_alloc_fail_error, Bus_Fxl.fail_error, Bus.fail_error>,
  ...
edon
node Bus_Fxl
 evt
  fail_error;
 sync
  <fail_error : COM.fail_error or MON.fail_error
                or S1.fail_error or S2.fail_error>,
 ...
edon
...
```

En effet la synchronisation des défaillances des différentes ressources manipulant les données véhiculées par ce bus, nous ramène au cas du double erroné présenté dans la section 3.2, cette situation n'étant pas acceptable pour une seule défaillance, nous pouvons en déduire que cette architecture proposée ne suffit pas pour tenir cette exigence.

Après cette première constatation, essayons avec une deuxième architecture constituée de deux ressources de communication.

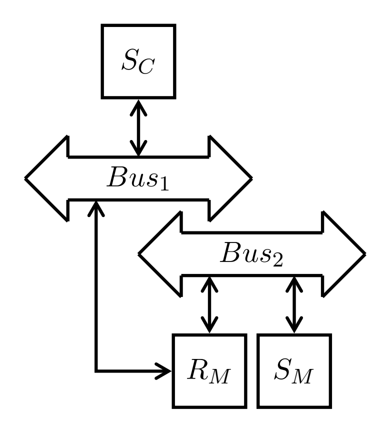


Fig. 3.21 – Deuxième Architecture pour le COM/MON

Parmi les ressources disponibles dans l'architecture matérielle, on remarque que les ressources S_M et R_M sont connectées aux deux Bus. Par conséquent, afin de suivre les directives, l'allocation de la voie du composant `Interrup` se fera sur la ressource S_C et l'allocation du composant `Equals` se fera sur la ressource S_M. Les autres allocations se feront sur les ressources restantes à savoir : S_1 et COM sont alloués sur la ressource R_C, S_2 et MON sont alloués sur S_M.

Une représentation possible de cette allocation est faite par la figure suivante :

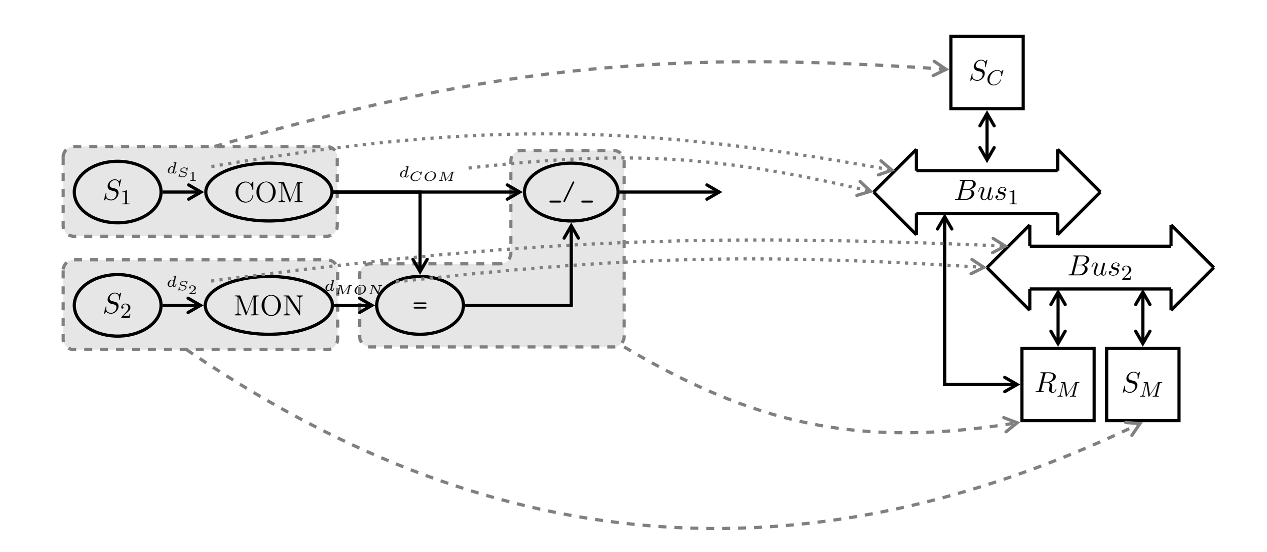


Fig. 3.22 – Une allocation possible pour le COM/MON

La modélisation AltaRica correspondant à cette allocation est représentée par l'ajout des synchronisations correspondantes dans le noeud englobant à la fois l'architecture matérielle et l'architecture fonctionnelle.

Le code AltaRica permettant la vérification de cette allocation est le suivant :

```
node Main
 sub
  S1, S2 : source,
  COM,MON : fonct,
  Equals : equals,
  Interrup : interrup;
  SC, RC, RM, SM, Bus1, Bus2: ress
 event
  SC_alloc_fail_lost, RM_alloc_fail_lost, ...
 assert
 ...
 sync
  <SC_alloc_fail_lost, SC.fail_lost, S1.fail_lost, COM.fail_lost>,
  <SM_alloc_fail_lost, SM.fail_lost, S2.fail_lost, MON.fail_lost>,
  <RM_alloc_fail_lost, RM.fail_lost, Equals.fail_lost, Interrup.fail_lost>,
  ...
edon
```

Les nouvelles analyses effectuées sur ce modèle permettent d'illustrer qu'aucune panne unique ne mène à la situation du double erroné. En effet, parmi les scénarios minimaux menant à la situation redoutée d'une propagation par l'interrupteur d'une donnée erronée, l'allocation ne rajoute que 6 scénarios et aucun n'est à l'ordre 1.

```
'COM.fail_error' & 'SM_alloc_fail_error';
'S1.fail_error' & 'SM_alloc_fail_error';
'Equals.fail_error' & 'SC_alloc_fail_error';
'MON.fail_error' & 'SC_alloc_fail_error';
'S2.fail_error' & 'SC_alloc_fail_error';
'SC_alloc_fail_error' & 'SM_alloc_fail_error';
```

Par cette constatation, l'architecture matérielle devient acceptable.

3.5 Bilan

Dans ce chapitre, nous avons tout d'abord montré que le langage AltaRica permet de décrire les architectures fonctionnelles et matérielles ainsi qu'un modèle global représentant une allocation de l'architecture fonctionnelle sur l'architecture matérielle. C'est sur ce dernier que des analyses de sûreté de fonctionnement peuvent être établies afin de prendre en compte l'impact des pannes de l'architecture matérielle sur les fonctionnalités du système.

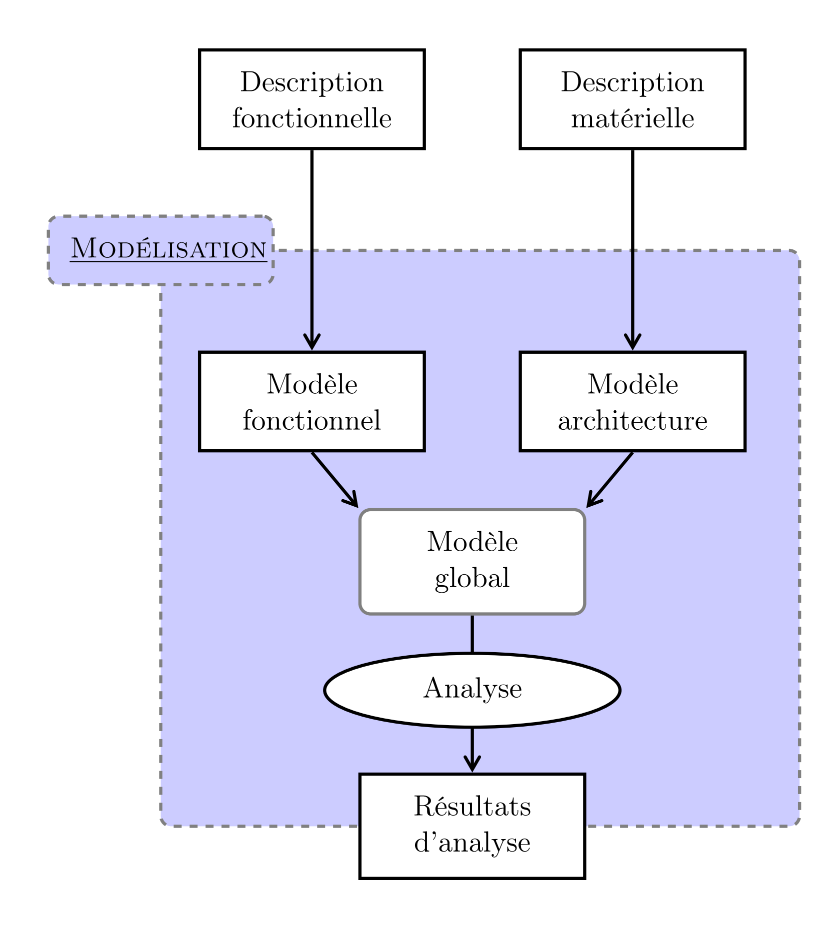


Fig. 3.23 – Démarche pour la modélisation

Nous avons montré sur l'exemple simple du COM/MON que le choix de cette allocation détermine la satisfaction des exigences du système. Dans cet exemple, le nombre restreint d'exigences et de composants des architectures autorise l'obtention manuelle d'une allocation satisfaisante. Cependant, déterminer une allocation dans le cadre d'un système plus complexe s'avère être une tâche difficilement réalisable et sans aucune garantie de résultat. C'est pourquoi nous proposons dans le chapitre suivant, une méthode de définition automatique d'allocation basée sur le principe de résolution de contraintes.

Chapitre 4

Résolution de contraintes d'allocation

Comme vu précédemment, la validation d'une allocation passe par l'analyse des architectures et la vérification des exigences suivant l'allocation. Nous souhaitons pouvoir aborder le problème d'une manière opposée, c'est-à-dire partant d'une description de l'architecture fonctionnelle et des différentes exigences à vérifier, chercher à en déduire une relation d'allocation ainsi que l'architecture matérielle qui les garantit.
Dans cette partie, nous détaillons comment utiliser le formalisme des systèmes de contraintes pour résoudre un problème d'allocation.
Le problème d'allocation consiste à trouver l'allocation *alloc* respectant un ensemble de contraintes (énumérées par la suite). La résolution de ce problème est possible à condition de transformer le problème d'allocation en un Problème de Satisfaction de Contraintes (CSP : *Constraint Satisfaction Problem*).

Sommaire

4.1 Modélisation et approche par contraintes

Définition 4.1 : Programmation par contraintes

La programmation par contraintes consiste à décrire un problème en termes de variables et de contraintes à satisfaire. Chaque contrainte exprime une propriété devant être vérifiée par un sous ensemble de variables. Une contrainte est une condition logique sur un ensemble de variables.

La modélisation d'un problème de satisfaction de contrainte ou CSP (*Constraint Satisfaction Problem*) consiste à identifier les variables et leurs domaines de définition représentant les ressources et les fonctions du problème, ainsi que l'ensemble des contraintes portant sur ces variables.

Définition 4.2 : CSP

Un CSP *est modélisé comme un triplet* $(X,\ D,\ C)$ *tel que :*

- X *est un ensemble de variables ;*
- D *est un ensemble de domaines tel qu'un domaine* $D(x)$ *soit associé avec chaque* $x \in X$.
- C *est un ensemble de contraintes sur les variables de* X.

Définition 4.3 : Résolution d'un CSP

Résoudre un CSP *consiste à trouver une ou toutes les solutions possibles (autorisées par les contraintes).*

Le problème de satisfaction de contraintes (CSP) consiste à trouver une affectation de valeurs à l'ensemble des variables (appelée instanciation des variables) telle que toutes les contraintes soient satisfaites.

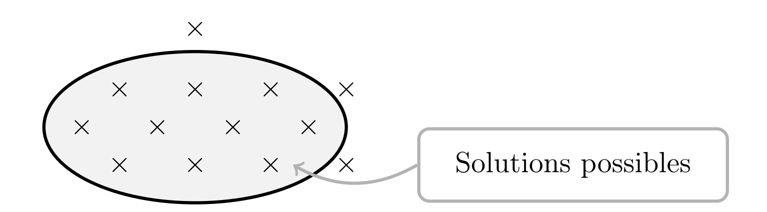


FIG. 4.1 – Solutions respectant les contraintes

Exemple de programmation par contraintes :

Prenons l'exemple de CSP connu du placement de plusieurs reines sur un échiquier (que l'on pourrait transcrire en un problème d'allocation de reines sur un échiquier tout en respectant un ensemble d'exigences) : Le problème des $\mathcal{N}$-Reines consiste à placer n reines sur un échiquier $(n \times n)$ sans qu'elles se menacent (cf. figure 4.2). Les contraintes associées à ce problème sont que deux reines ne doivent pas se trouver sur la même ligne, sur la même colonne ou sur la même diagonale.

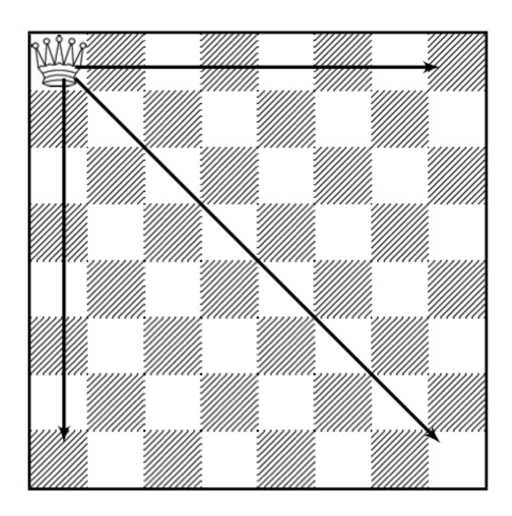


Fig. 4.2 – Problème des $\mathcal{N}$-Reines

Une modélisation possible du problème consiste à associer pour chacune des cases une variable prenant la valeur 1 si la reine est présente sur la case et la valeur 0 sinon. On notera X_{ij} la variable qui représente la case correspondante à la ligne i et à la colonne j de l'échiquier. Les contraintes spécifient qu'il ne peut y avoir plus d'une reine sur une même ligne, sur une même colonne ou sur une même diagonale.

Le CSP de ce problème devient le suivant :

– les variables : $X = \{X_{ij} | i$ le numéro de ligne et j le numéro de colonne$\}$
– le domaine $D = \{0, 1\}$
– les contraintes :

1. Une seule reine par ligne : $\forall i \in \{1 \ldots n\}, \sum_{j=1}^{n} X_{ij} = 1$
2. Une seule reine par colonne : $\forall j \in \{1 \ldots n\}, \sum_{i=1}^{n} X_{ij} = 1$
3. Deux reines ne doivent pas se retrouver sur une même diagonale :

$$\forall i, j, k, l \in \{1 \ldots n\}, X_{ij} + X_{kl} \leq 1 \text{ avec } |i - k| = |j - l|$$

La résolution de ce CSP lorsque n vaut 8 (c.-à-.d dans le cas d'un échiquier de 8 lignes) donne 92 affectations possibles et différentes, dont voici un exemple :

$$(X_{11}, X_{25}, X_{38}, X_{46}, X_{53}, X_{67}, X_{72}, X_{84})$$

Cette solution est représentée par la figure 4.3.

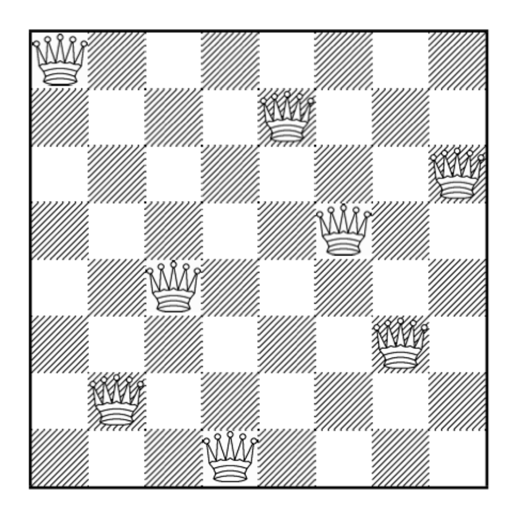


Fig. 4.3 – Exemple de solution du problème des $\mathcal{N}$-Reines

4.2 Les variables du problème d'allocation

La modélisation du problème d'allocation passe par une première étape d'identification des différentes variables. Les variables utilisées par la modélisation sont les suivantes :

Les ensembles : Afin de manipuler les différents composants, nous les avons regroupés en deux ensembles : *Fonction* ($\mathcal{F}$) et *Ressource* ($\mathcal{R}$).

Les constantes : Différentes relations peuvent s'appliquer sur des fonctions :

- f_cnx correspond à la relation de connexion entre les fonctions définies dans la section 3.1. Pour rappel, si $f_cnx(f_1, f_2) = 1$ alors f_1 est connecté à f_2

- *coloc* permet de préciser que deux fonctions doivent se retrouver allouées sur une même ressource.

$$\begin{aligned} coloc : \ & \mathcal{F} \times \mathcal{F} \rightarrow \{0, 1\} \\ & coloc(f_1, f_2) = 1 : f_1 \text{ et } f_2 \text{ sont à placer sur une même ressource} \end{aligned}$$

- *idpt* correspond à la relation d'*indépendance* entre deux fonctions. Elle permet de préciser que deux fonctions ne doivent pas se retrouver allouées sur une même ressource.

$$\begin{aligned} idpt : \ & \mathcal{F} \times \mathcal{F} \rightarrow \{0, 1\} \\ & idpt(f_1, f_2) = 1 : f_1 \text{ et } f_2 \text{ sont à placer sur des ressources différentes} \end{aligned}$$

- *set_alloc* correspond pour une fonction donnée à l'ensemble des ressources autorisant une allocation.

$$\begin{aligned} set_alloc : \ & \mathcal{F} \rightarrow \mathcal{P}(\mathcal{R}) \\ & set_alloc(f_1) = \{r_1, \ldots, r_n\} : \ f_1 \text{ doit être allouée à une ressource parmi } r_1, \ldots, r_n. \end{aligned}$$

Les variables :

- u_cnx est contenu dans la fermeture réflexive, symétrique et transitive de la relation r_cnx et correspond à la relation de connexion entre ressources utilisées. Ainsi, lorsque $u_cnx(r_1, r_2) = 1$ alors r_1 est connecté à r_2. Ceci signifie que nous ne considérons pas forcément que l'interconnexion des ressources de l'architecture matérielle est figée. Nous recherchons les connexions qui doivent être utilisées.

– *alloc* est la relation que nous souhaitons établir entre les ressources et les fonctions.

$$alloc: \quad \mathcal{F} \times \mathcal{R} \rightarrow \{0,1\}$$
$$alloc(f,r) = 1$$

4.3 Le domaine des variables utilisées

Comme présenté précédemment lors de la déclaration des différentes variables, le domaine des valeurs est l'ensemble des booléens. En effet, nous souhaitions un moyen simple pour exprimer une relation définie pour chaque variable. Par exemple, étant donné que le problème qui nous intéresse est un problème d'allocation, l'utilisation de variables booléennes permet d'affirmer pour chaque fonction si oui ou non elle est allouée à une ressource : $alloc(f_1, r_1) = 0$ signifiant que la fonction f_1 n'est pas allouée sur la ressource r_1.

De plus, l'avantage de la modélisation à partir de variables booléennes est que cela permet de construire facilement notre problème de contraintes comme un problème SAT[1]. En effet, un problème SAT consiste à déterminer si une formule construite à partir de variables booléennes admet ou non une solution, c.-à-d. s'il existe une attribution de valeur à chacune des variables qui satisfait la formule. Nous verrons dans la suite comment construire ce problème SAT à partir des contraintes.

4.4 Les contraintes

Les différentes contraintes utiles à notre modélisation peuvent être regroupées en 2 catégories : des contraintes sur les connexions autorisées et des contraintes sur le placement des fonctions sur les ressources.

Ces contraintes sont ensuite enrichies par des directives d'allocations issues des choix de conception.

Les contraintes de placement : Elles concernent le positionnement des fonctions sur l'ensemble des ressources.

1. « *Toutes les fonctions doivent être allouées* »

   Cette contrainte est due à la nécessité pour une tâche d'utiliser une ressource matérielle afin de pouvoir s'exécuter. En effet, la modélisation de l'architecture fonctionnelle correspond à la mise en relation de l'ensemble des fonctions nécessaires au système avec l'ensemble des ressources mises à disposition. Chacune des fonctions modélisées a un rôle important pour le système global, il faut donc s'assurer que lors de l'allocation, aucune fonction ne se retrouve orpheline (c.-à-.d qu'il n'existe aucune fonction non allouée).

$$\forall f \in \mathcal{F}, \exists r \in \mathcal{R} : alloc(f,r) = 1 \tag{4.1}$$

2. « *Une fonction ne doit être allouée qu'à une seule ressource* »

   Parmi les choix de modélisation, il a été décidé que toutes les fonctions étaient « uniques ». Il faut comprendre par unique le fait que chaque fonction est dédiée à une tâche bien spécifique. Nous considérons par exemple que la redondance d'une fonction (technique fréquemment utilisée pour la tolérance aux fautes) engendre 2 fonctions différentes mais possédant exactement le même comportement fonctionnel. Ces choix de modélisation permettent de n'avoir qu'une seule ressource à utiliser pour chaque fonction.

$$\forall f \in \mathcal{F}, \forall r_1, r_2 \in \mathcal{R} : alloc(f,r_1) \wedge alloc(f,r_2) \Rightarrow r_1 = r_2 \tag{4.2}$$

[1] Boolean satisfiability problem

Les contraintes de connexions :

« *Deux fonctions connectées doivent le rester une fois allouées* ».

Cette contrainte stipule que l'allocation doit conserver les connexions entre les fonctions. En effet, si lors de la modélisation de l'architecture fonctionnelle plusieurs fonctions sont amenées à communiquer entre elles, il faut qu'une fois allouées elles puissent continuer à le faire. Pour cela, il faut s'assurer que lors de l'allocation, les ressources hébergeant les fonctions restent bien en connexion (cf. figure 4.4).

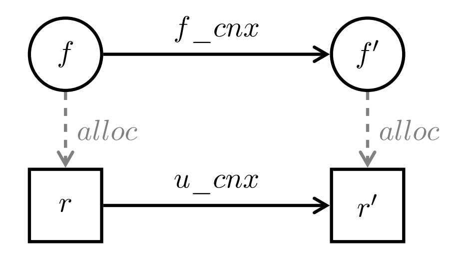


Fig. 4.4 – Contrainte de connexion

Pour permettre cela, il faut supposer que la relation : u_cnx respecte deux propriétés :la transitivité et la réflexivité.

En effet, lorsque plusieurs fonctions interconnectées sont allouées sur une même ressource, nous souhaitons pouvoir conserver la propriété de cette interconnexion. Ainsi la propriété de réflexivité de u_cnx permet de spécifier que :$\forall r \in \mathcal{R} : (r, r) \in u_cnx$ (cf. figure 4.5)

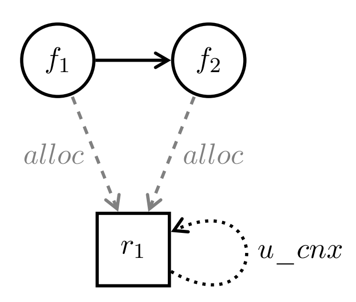


Fig. 4.5 – u_cnx relation réflexive

La propriété de transitivité permet d'exprimer par une simple relation entre deux ressources le fait qu'il existe un chemin entre ces deux ressources. Un chemin correspond à une route entre deux ressources en utilisant des connexions avec d'autres ressources. La transitivité de u_cnx signifie donc dans notre problème que :

$$\forall r_1, r_2, r_3 \in \mathcal{R} : \left\{ \begin{array}{l} (r_1, r_2) \in u_cnx \\ (r_2, r_3) \in u_cnx \end{array} \right. \Rightarrow (r_1, r_3) \in u_cnx$$

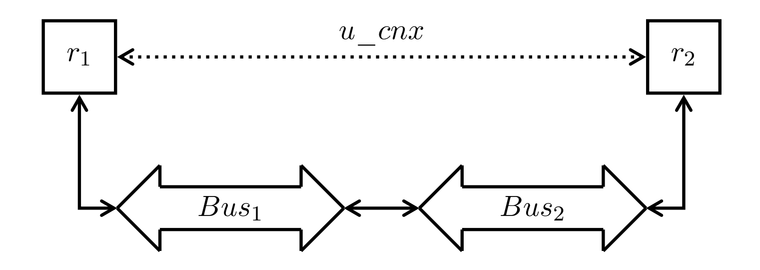


Fig. 4.6 – u_cnx fermeture transitive

Ainsi, la contrainte sur les connexions peut s'écrire en logique de façon suivante :

$$\forall f, f' \in \mathcal{F}, \forall r, r' \in \mathcal{R} : alloc(f, r) \wedge alloc(f', r') \wedge f_cnx(f, f') \Rightarrow u_cnx(r, r') \tag{4.3}$$

Les directives d'allocations : Elles concernent les choix prédéfinis par le concepteur du système.

1. **Set-allocation** : « t_1 *doit être allouée sur une ressource possédant certaines caractéristiques* »

   Suivant l'importance de la fonction, il est interdit de l'allouer à des ressources qui ne posséderaient pas des caractéristiques suffisantes pour une exécution idéale. En effet, certaines fonctions critiques comme le pilotage automatique nécessitent des ressources garantissant un certain niveau de sécurité. Il est indispensable lors de la conception de système de bien respecter les caractéristiques nécessaires aux fonctions pour un bon fonctionnement afin de garantir les performances et les exigences. Afin de faciliter les différents choix offerts aux concepteurs, la contrainte *Set-allocation* permet de spécifier les ressources nécessaires pour chaque fonction. Elle permet ainsi de réduire les possibilités d'allocation et aussi de garantir quelques exigences comme la performance et la sécurité.

   Par exemple, prenons une architecture fonctionnelle composée de quatre fonctions (f_1, f_2, f_3, f_4) connectées entre elles, et pour permettre l'allocation, l'architecture matérielle proposée est composée de trois ressources (r_1, r_2, r_3) ayant des caractéristiques différentes. Afin d'orienter les possibilités d'allocations, les directives suivantes sont proposées :
   - la fonction f_1 ne peut s'exécuter que sur une ressource de type r_1, et
   - la fonction f_3 quant à elle ne doit être allouée que sur une ressource ayant les mêmes caractéristiques que r_2 et r_3.

   Pour modéliser ces directives, il faut définir $set_alloc(f_1) = \{r_1\}$ et $set_alloc(f_2) = \{r_2, r_3\}$.
   Ces directives peuvent être représentées par la figure suivante :

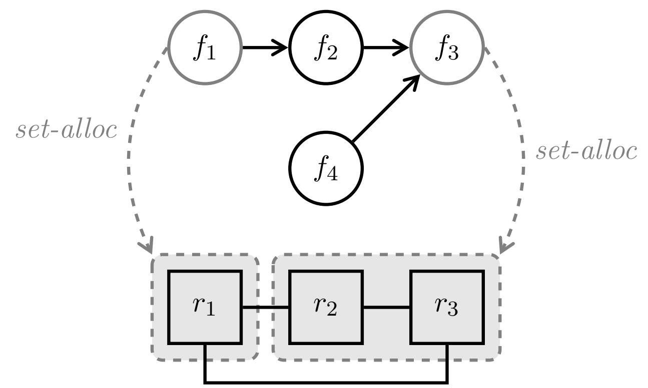


FIG. 4.7 – Illustration de la relation *Set-Allocation*

La contrainte associée à cette directive doit s'assurer que si l'allocation d'une fonction sur une ressource est proposée alors la ressource doit appartenir à l'ensemble des ressources permettant son exécution.

Ainsi lorsque l'ensemble $set_alloc(f_i)$ est défini, au moins une allocation doit être effectué sur une ressource de cet ensemble. Cette contrainte peut s'écrire en logique de façon suivante :

$$\forall f_i, \in \mathcal{F}, \exists r \in set_alloc(f_i) : alloc(f_i, r) \tag{4.4}$$

2. **Co-localisation** : « t_1 *et* t_2 *doivent être allouées sur une même ressource* »

   Suivant l'architecture disponible et suivant les systèmes, il se peut que certaines fonctions soient amenées à communiquer entre elles fréquemment. Sachant que ces fonctions sont amenées à s'échanger des données, il semble logique d'éviter que ces données aient un long chemin à parcourir (on entend par chemin le parcours des données dans les différentes ressources matérielles nécessaires pour connecter ces fonctions). L'action de rapprocher les ressources hébergeant ces fonctions ou bien le fait de choisir l'allocation de ces fonctions sur une même ressource est important pour le concepteur. Afin de simplifier cette démarche, une nouvelle contrainte permet de spécifier les différents regroupements de fonctions possibles à allouer sur une même ressource.

   La contrainte correspondante pour imposer l'allocation de deux fonctions sur une même ressource est la suivante :

$$\forall f_1, f_2 \in \mathcal{F}, \forall r \in \mathcal{R} : coloc(f_1, f_2) \Rightarrow (alloc(f_1, r) \Leftrightarrow alloc(f_2, r)) \tag{4.5}$$

3. **Allocation** : L'imposition d'une certaine allocation à une fonction f peut se traduire en une affectation de la variable correspondante ($alloc(f, r_i)$). Par conséquent, la contrainte permettant d'imposer le choix de l'allocation d'une fonction f sur une ressource r s'exprime de la façon suivante :

$$alloc(f, r) = 1$$

4. **Exclusion** :
   A l'inverse, certaines ressources peuvent ne pas avoir un degré de fiabilité suffisant pour autoriser l'allocation d'une fonction dite « critique ». En effet lorsque l'architecture matérielle le permet, plusieurs ressources ayant des caractéristiques différentes peuvent être mises à disposition. Suivant l'importance et le degré de fiabilité de la fonction, le concepteur peut souhaiter interdire certaines ressources n'ayant pas les caractéristiques suffisantes pour exécuter certaines fonctions.

La contrainte permettant l'exclusion de certaines fonctions sur des ressources est exactement l'opposé de la contrainte de la précédente, et s'écrit de la façon suivante :

$$alloc(t_2, r_2) = 0$$

5. **Indépendance** : « f_1 *ne doit pas être allouée sur la même ressource que* f_2 » :

   Cette contrainte traite l'indépendance de deux fonctions.

   Deux fonctions sont considérées comme indépendantes lorsqu'elles ne doivent pas se retrouver allouées sur une même ressource. Cette indépendance est nécessaire afin de respecter certaines exigences de sûreté de fonctionnement. Sachant que ces exigences visent à contrôler le nombre de pannes menant à une situation redoutée, lorsque celle-ci concerne une panne simple, il ne faut pas qu'une seule défaillance sur une ressource impacte plusieurs fonctions et par conséquent invalide l'exigence. D'où l'importance de respecter l'indépendance lorsqu'elle est clairement identifiée et d'isoler certaines fonctions importantes pour le système.
   La contrainte correspondante pour empêcher l'allocation de deux fonctions indépendantes sur une même ressource est la suivante :

$$\forall f_1, f_2 \in \mathcal{F}, \forall r \in \mathcal{R} : idpt(f_1, f_2) \Rightarrow \neg(alloc(f_1, r) \wedge alloc(f_2, r)) \quad (4.6)$$

4.5 Transformation du modèle csp en système d'équations linéaires

Nous avons, jusque là, décrit notre problème d'allocation par un ensemble de variables et des contraintes reliant ces variables entre elles. Les différentes allocations « acceptables » sont représentées par les différentes combinaisons de valeurs des variables qui sont compatibles avec les contraintes.

L'affectation des valeurs aux différentes variables est possible en faisant appel à un outil de résolution de contraintes.

Une fois notre problème modélisé sous la forme d'un csp, l'étape suivante consiste à traduire les constituants du modèle en éléments de base formant notre système d'équations linéaires. Ce modèle a été transformé (moyennant l'ajout de variables booléennes appropriées) en modèle de programmation linéaire booléen.

4.5.1 Des relations vers des variables booléennes

Comme précisé dans la section précédente, l'outil SatZoo[ES03, GT04] qui permet de résoudre notre csp, prend comme variables d'entrées des variables booléennes. Or comme présenté dans la première section, les variables utilisées pour la modélisation sont des relations. La transformation en variables booléennes consiste à lister toutes les variables lorsqu'une relation est vraie.

La relation f_cnx permettant d'exprimer la connexion entre deux fonctions peut se transformer en un ensemble de variables booléennes exprimant les différentes connexions. Cette transformation permet aussi de construire l'ensemble $\mathcal{V}_{u_cnx}$ correspondant aux connexions entre les ressources.

Considérons par exemple le problème d'allocation de 4 fonctions (f_1, f_2, f_3, f_4) connectées comme d'après la figure 4.8 sur un ensemble de 3 ressources supposées interconnectées.

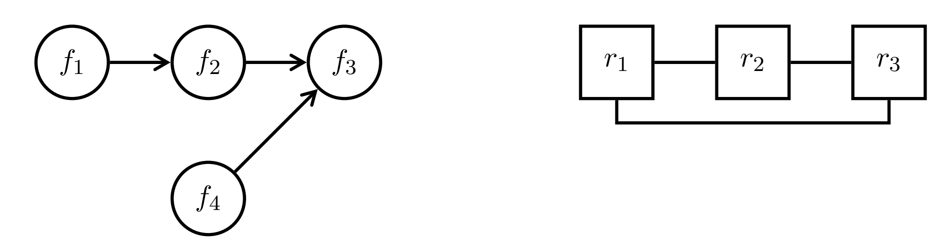


Fig. 4.8 – Exemple de problème d'allocation

Il existe ici des connexions entre les fonctions (f_1 avec f_2, f_2 avec f_3 et f_4 avec f_3). La relation f_{cnx} correspondante à cet exemple est la suivante :

$$\begin{cases} f_cnx(f_1, f_2) = 1 \\ f_cnx(f_2, f_3) = 1 \\ f_cnx(f_4, f_3) = 1 \end{cases}$$

L'ensemble des variables booléennes nécessaires est donc le suivant :

$$\mathcal{V}_{f_cnx} = \{ \texttt{v_f_cnx_f1_f2,v_f_cnx_f2_f3 ,v_f_cnx_f3_f4} \}$$

Une fois toutes les variables identifiées, il faut adapter nos différentes contraintes afin de construire notre système d'équations linéaires.

4.5.2 Des variables booléennes vers des inéquations linéaires

Nous définissons des lois de transformations $trad()$ applicables à des formules en forme normale conjonctive, c'est à dire de la forme :

$$\Phi \equiv \bigwedge_{i \in 1 \ldots k} Clause_i \text{ où } \begin{cases} Clause_i = (\varphi_1 \vee \ldots \vee \varphi_n) \\ \varphi_j \text{ est un littéral} \end{cases}$$

Soit φ une variable alors $trad(\varphi) = v_\varphi$.

Afin d'évaluer nos lois de transformation, la comparaison des tables de vérité s'avère nécessaire (φ variable).

φ	$\neg\varphi$	$1 - v_\varphi$
0	1	1
1	0	0

Ces deux colonnes étant identiques, nous pouvons en déduire une première loi de transformation sur la négation :

$$trad(\neg\varphi) = (1 - v_\varphi)$$

Intuitivement,une formule disjonctive ($\varphi \vee \psi$) doit pouvoir se traduire en la somme des éléments avec une inégalité. Vérifions cette intuition en comparant les tables ($\varphi \vee \psi$ littéraux) :

φ	ψ	$\varphi \vee \psi$	$v_\varphi + v_\psi$	$v_\varphi + v_\psi \geq 1$
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	1	2	1

A nouveau, la comparaison des tables permet de valider cette règle de transformation pour les formules disjonctives :

$$trad(\varphi \vee \psi) = ((v_\varphi + v_\psi) \geq 1)$$

Les différentes règles de transformation que nous venons d'identifier nous permettent de passer d'une contrainte décrite en logique à un système d'inéquations linéaires. Elles sont résumées dans le tableau suivant (φ,ψ littéraux) :

$\varphi, \psi : vrai$	$\Leftrightarrow$	$v_\varphi \geq 1, v_\psi \geq 1$
$\neg\varphi : vrai$	$\Leftrightarrow$	$1 - v_\varphi \geq 1$
$\varphi \vee \psi : vrai$	$\Leftrightarrow$	$v_\varphi + v_\psi \geq 1$

L'application de ces différentes règles de transformations sur nos contraintes « logiques » permet d'obtenir les inéquations linéaires correspondantes suivantes :

$$\begin{array}{lcl} \Phi = \bigwedge_{i \in \mathcal{I}} Clause_i \text{ vrai} & \Leftrightarrow & \forall i \in \mathcal{I} : \; trad(Clause_i) \geq 1 \\ Clause_i = (\varphi_i^1 \vee \ldots \vee \varphi_i^n) & \Leftrightarrow & \sum_{j=1}^{n} trad(\varphi_i^j) \geq 1 \end{array}$$

- « *Deux fonctions connectées doivent le rester une fois allouées* » (4.3) :
  Partant de la formule logique définie précédemment, il faut trouver sa forme normale conjonctive correspondante :
  $\forall f, f' \in \mathcal{F}, \forall r, r' \in \mathcal{R}$:

$$\begin{array}{ll} & alloc(f,r) \wedge alloc(f',r') \wedge f_cnx(f,f') \Rightarrow u_cnx(r,r') \\ \Leftrightarrow & \neg(alloc(f,r) \wedge alloc(f',r') \wedge f_cnx(f,f')) \vee u_cnx(r,r') \\ \Leftrightarrow & \neg alloc(f,r) \vee \neg alloc(f',r') \vee \neg f_cnx(f,f') \vee u_cnx(r,r') \end{array}$$

  Comme cette dernière formule est en forme normale conjonctive, on peut lui appliquer les règles de transformation et ainsi obtenir :

$$\begin{array}{ll} & \neg alloc(f,r) \vee \neg alloc(f',r') \vee \neg f_cnx(f,f') \vee u_cnx(r,r') : \text{vrai} \\ \Leftrightarrow & 1 - alloc(f,r) + 1 - alloc(f',r') + 1 - f_cnx(f,f') + u_cnx(r,r') \geq 1 \\ \Leftrightarrow & 3 - alloc(f,r) - alloc(f',r') - f_cnx(f,f') + u_cnx(r,r') \geq 1 \\ \Leftrightarrow & alloc(f,r) + alloc(f',r') + f_cnx(f,f') - u_cnx(r,r') \leq 2 \end{array}$$

  Ainsi, la contrainte pouvant s'intégrer dans notre système d'équations linéaires est :

$$\begin{array}{l} \forall f, f' \in \mathcal{F}, \forall r, r' \in \mathcal{R} : \\ alloc(f,r) + alloc(f',r') + f_cnx(f,f') - u_cnx(r,r') \leq 2 \end{array}$$

- « *Toutes les fonctions doivent être allouées* » (4.1) et
  « *Une fonction ne peut être allouée qu'à une seule ressource* » (4.2), ces contraintes permettent d'obtenir une nouvelle contrainte pour enrichir le système.

$$\forall r \in \mathcal{R} : \sum_{f_i \in \mathcal{F}} alloc(f_i, r) = 1$$

- Les traductions des contraintes sur les directives d'allocation sont :
  1. *set-allocation* (4.4) :

$$\forall f \in \mathcal{F} : \sum_{r_i : set_alloc(f)} alloc(f, r_i) = 1$$

  2. *co-localisation* (4.5)

$$coloc(f_1, f_2) \Rightarrow (alloc(f_1, r) \Leftrightarrow alloc(f_2, r))$$

$$\Leftrightarrow \left\{ \begin{array}{l} coloc(f_1, f_2) \Rightarrow (alloc(f_2, r) \Rightarrow alloc(f_1, r)) \\ coloc(f_1, f_2) \Rightarrow (alloc(f_1, r) \Rightarrow alloc(f_2, r)) \end{array} \right.$$

  La traduction de cette contrainte revient donc à traduire ces deux sous-contraintes et la traduction de la première sous-contrainte en inéquation suffit pour identifier la deuxième inéquation correspondante.

$$\begin{array}{ll} & coloc(f_1, f_2) \Rightarrow (alloc(f_1, r) \Rightarrow alloc(f_2, r)) \\ \Leftrightarrow & \neg coloc(f_1, f_2) \vee \neg alloc(f_1, r) \vee alloc(f_2, r) : \text{vrai} \\ \Leftrightarrow & 1 - coloc(f_1, f_2) + 1 - alloc(f_1, r) + alloc(f_2, r) \geq 1 \\ \Leftrightarrow & 2 - coloc(f_1, f_2) + 1 - alloc(f_1, r) + alloc(f_2, r) \geq 1 \\ \Leftrightarrow & coloc(f_1, f_2) + alloc(f_1, r) - alloc(f_2, r) \leq 1 \end{array}$$

Ainsi, les contraintes à satisfaire sont :

$$\forall f_i, f_j \in \mathcal{F}, \forall r \in \mathcal{R} : \begin{cases} coloc(f_1, f_2) + alloc(f_1, r) - alloc(f_2, r) \leq 1 \\ coloc(f_1, f_2) + alloc(f_2, r) - alloc(f_1, r) \leq 1 \end{cases}$$

3. *indépendance* (4.6) :

$$\forall f_i, f_j \in \mathcal{F}, \forall r \in \mathcal{R} : alloc(f_i, r) + alloc(f_j, r) + idpt(f_i, f_j) \leq 2$$

4.6 Résolution du système de contraintes

Le nombre de contraintes et de variables dépend du nombre de composants utilisés dans les architectures fonctionnelle et matérielle. Or, plus le nombre de variables est grand, plus la satisfaction de contraintes peut prendre du temps et plus la solution s'avère difficile à trouver. En raison du problème d'explosion combinatoire dû au fait que le problème appartient à la classe des problèmes NP-complet [Coo71], il est préférable de limiter le nombre de variables mais aussi le nombre de contraintes pour obtenir une allocation possible.

Ainsi, les contraintes sur les différents choix d'allocation (imposer ou interdire l'allocation d'une fonction sur une ressource) sont directement appliquées lors de la déclaration des variables pour réduire les différentes combinaisons possibles. En effet, lors de la déclaration des variables nécessaires à notre problème, il est possible d'instancier certaines variables afin de limiter les possibilités d'allocations. Ainsi les directives d'allocations *allocation* et *exclusion* sont directement appliquées lors de l'instanciation des variables. Elles permettent ainsi de minimiser certaines variables d'allocation du système en les transformant en constantes.

Nous avons développé un outil qui, partant d'une description de l'architecture fonctionnelle (et de l'architecture matérielle lorsque celle-ci est disponible) du modèle, produit un ensemble de contraintes linéaires entières, pouvant être traitées par un solveur de contraintes (par exemple SatZoo). Un descriptif détaillé du fonctionnement de l'outil est donné par la figure 4.6.

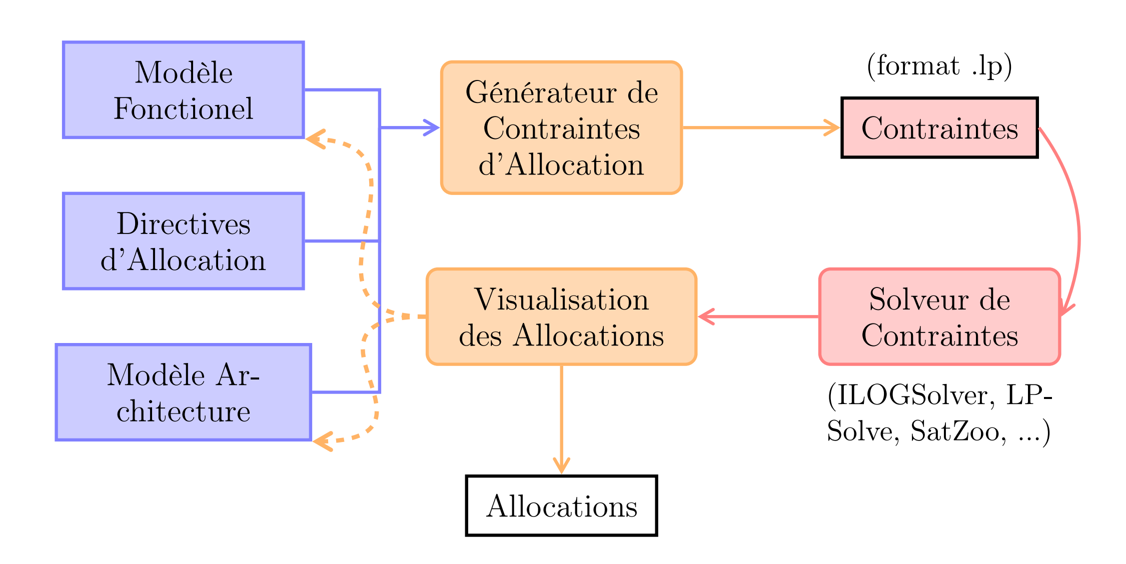


Fig. 4.9 – Outil pour la recherche et la visualisation d'allocations

Voyons à présent comment cet outil fonctionne sur l'exemple du COM/MON du chapitre précédent. Les informations nécessaires et suffisantes à extraire du modèle pour construire le système sont :

- les connexions entre ses composants,
- l'indépendance entre les composants,
- les différents choix possibles pour orienter des allocations.

Dans un premier temps, ces informations vont permettre la valuation des constantes du système (respectivement *f_ cnx*, *idpt*, *coloc* et *set_ alloc*). Un exemple d'informations à extraire de l'exemple du COM/MON est donné par la figure suivante (les informations à extraire et permettant de construire le système de contraintes sont décrites dans le cadre grisé) :

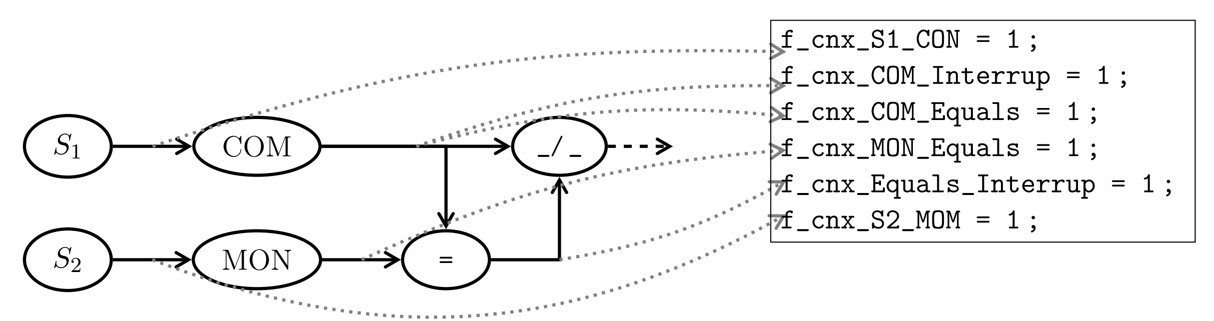


Fig. 4.10 – Description du modèle *COM/MON*

Étant donné que les ressources de calcul possèdent les mêmes caractéristiques, nous pouvons considérer que toutes les fonctions du COM/MON peuvent être allouées sur l'ensemble des ressources mises à disposition. L'architecture matérielle que nous allons utiliser est la même que celle présentée dans le chapitre précèdent, à savoir trois ressources de calcul (R_M,S_M et S_C) et deux ressources de communication (Bus_1 et Bus_2).

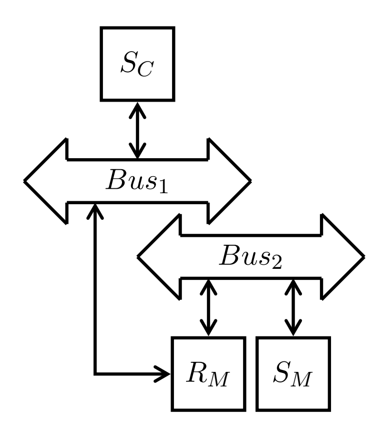


Fig. 4.11 – Architecture pour le COM/MON

Les directives d'allocation associées à l'architecture (voir section 4.4) apportent de nouvelles informations . Pour rappel, nous souhaitons que :

- Les couples ((S1,COM), (S2,MON) et (Interrup, Equals)) soient alloués sur une même ressource. Pour cela, nous ajoutons au système les contraintes suivantes :

```
coloc_S1_COM = 1;
coloc_S2_MON = 1;
coloc_Interrup_Equals = 1;
```

- Il y ait une indépendance entre COM et MON (et par conséquent il y ait aussi indépendance entre S1 et S2) :

```
idpt_COM_MON = 1;
idpt_S1_S2 = 1;
```

Partant de ce début de système, il suffit de le compléter avec les différentes contraintes pour ainsi obtenir le système d'équations linéaires. Voyons à présent comment construire les équations à partir des contraintes (nous ne traiterons pas l'ensemble des fonctions, mais nous nous focaliserons sur les équations construites à partir de la fonction COM) :

– Placement, set-allocation :

```
// COM allouée sur une seule ressource
alloc_COM_SM + alloc_COM_RM + alloc_COM_SC = 1;
```

– Indépendance :

```
// Pour l'indépendance entre COM et MON
alloc_COM_SM + alloc_MON_SM <= 1;
alloc_COM_RM + alloc_MON_RM <= 1;
alloc_COM_SC + alloc_MON_SC <= 1;
```

– Co-localisation :

```
// Pour la co-localisation de COM et S1
alloc_COM_SM - alloc_S1_SM <= 0;
- alloc_COM_SM + alloc_S1_SM <= 0;

alloc_COM_RM - alloc_S1_RM <= 0;
- alloc_COM_RM + alloc_S1_RM <= 0;

alloc_COM_SC - alloc_S1_SC <= 0;
- alloc_COM_SC + alloc_S1_SC <= 0;
```

– Connexion :

```
// Pour la liaison fonctionnelle entre S1 et COM
alloc_S1_SM + alloc_COM_RM- u_cnx_SM_RM <= 1;
alloc_S1_SM + alloc_COM_SC - u_cnx_SM_SC <= 1;
alloc_S1_RM + alloc_COM_SC - u_cnx_RM_SC <= 1;
```

Une fois le système d'équations linéaires créé, c.à.d. après réitération des traductions sur l'ensemble des fonctions, il suffit d'utiliser l'outil SATZOO afin qu'il nous propose une solution possible à notre problème d'allocation.

Une allocation proposée par l'outil SATZOO est la suivante :

```
u_cnx_RM_SC
u_cnx_RM_SM
alloc_COM_SC
alloc_MON_SM
alloc_S1_SC
alloc_S2_SM
alloc_Equals_RM
alloc_Interrup_RM
```

Cette allocation proposée peut facilement être représentée par la figure 4.12.

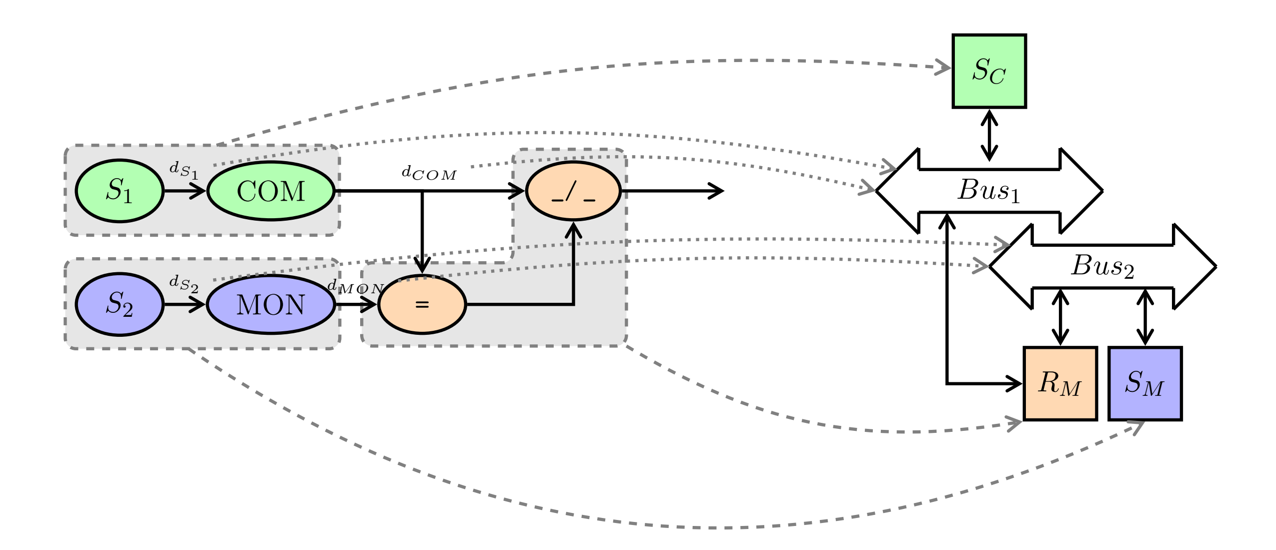


Fig. 4.12 – Allocation proposée par SatZoo

Cette allocation est tout à fait acceptable, car elle est identique à celle validée dans le chapitre précédent.

Cette démarche sera validée sur le problème de l'allocation d'un système de navigation (Suivi de Terrain) sur des ressources matérielles d'un avion militaire de type Dassault Mirage 2000 (cf. chapitre 6).

4.6.1 Trop d'allocations possibles ?

Plus nous ajoutons de contraintes et plus l'espace des solutions admissibles rétrécit. Mais dans certains cas, l'ensemble des solutions acceptables reste trop grand.

Un des principaux avantages à formaliser notre problème d'allocation en un problème de satisfaction de contraintes est qu'il est possible de spécifier un critère d'optimisation afin de classer les solutions possibles. Ce critère va permettre de choisir parmi l'ensemble des solutions possibles, celles qui semblent les plus intéressantes.

Durant notre étude, nous avons considéré un premier critère d'optimisation apportant un intérêt certain pour le concepteur. La solution optimale que nous allons considérer est une solution utilisant le minimum de ressources. Pour parvenir à cette solution optimale, nous allons présenter le critère d'optimisation utilisé.

Les critères d'optimisation Un critère d'optimisation correspond à un aspect global du système qui est exprimé en fonction de certaines variables. Rappelons que la résolution du système de contraintes correspond à la valuation des variables. Le critère consiste à maximiser ou minimiser la valeur d'un indicateur construit à partir de ces variables pour ainsi diminuer l'ensemble des solutions acceptables (cf. figure 4.5).

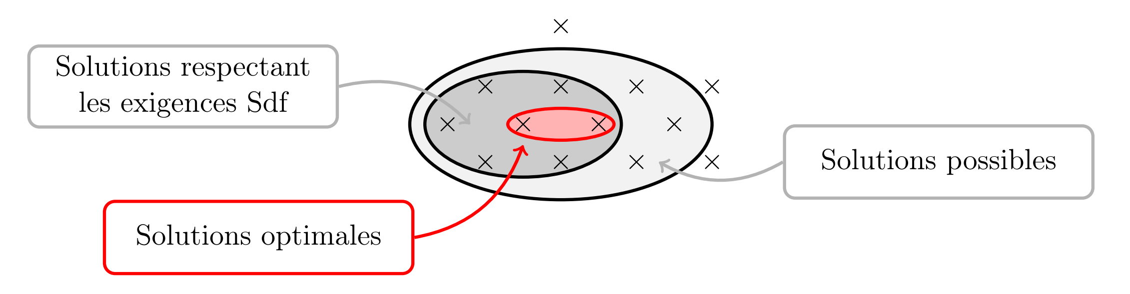


Fig. 4.13 – Ensemble des solutions optimales

L'ajout d'un critère à un système de contraintes s'effectue en plaçant la définition du critère en début du système (pour respecter le formalisme de l'outil SatZoo). Par exemple si nous souhaitons minimiser le nombre des variables v_1, v_2, v_3 qui sont vraies, il suffit d'écrire la définition suivante pour exprimer le critère :

$$z_min = \sum_{i=1}^{3} v_i$$

Minimiser le nombre de connexions entre les ressources L'utilisation de ce critère permet d'obtenir l'architecture optimale du point de vue du nombre de connexions utilisées. En effet, l'action de minimiser les connexions entre les ressources permet d'éliminer les connexions dites « inutiles ». Ces connexions sont qualifiées d'inutiles, car elles ne supportent aucun échange entre fonctions de l'architecture fonctionnelle. Lors de la conception d'un avion les concepteurs cherchent sans cesse de nouvelles technologies afin de limiter le nombre équipements et de câbles utilisés pour faire un gain d'espace et de poids au niveau de l'avion. Un critère alternatif serait de minimiser le nombre de ressources utilisées, mais il semble moins pertinent que celui que nous avons étudié. En effet, dans un avion, le poids et l'encombrement d'un calculateur sont négligeables devant le poids et l'encombrement des câbles permettant d'établir les connexions entre équipements. Par conséquent, en minimisant le nombre de connexions utilisées nous espérons proposer des architectures conduisant à un poids de câbles le plus faible possible. De plus, le nombre de ressources de calcul est souvent contraint et peut donc être pris en compte par notre recherche d'allocation comme une constante.

Le fait de supprimer automatiquement les connexions non utilisées permet de proposer aux concepteurs une solution possédant déjà de premières caractéristiques intéressantes.

Ainsi, le critère de minimisation du nombre de connexions s'écrit :

$$c_min = \sum_{r,r' \in \mathcal{R} \times \mathcal{R}} (u_cnx(r, r'))$$

L'application de ce critère à l'architecture COM/MON revient à rajouter au système l'équation suivante :

```
min + u_cnx_SC_SM + u_cnx_RM_SC + u_cnx_RM_SM ;
```

L'application de ce critère sur l'exemple du COM/MON va permettre de rechercher une allocation possédant le minimum de ressources de connexion nécessaires pour satisfaire l'ensemble des contraintes. En ajoutant ce critère au système d'équations linéaires créé précédemment, l'outil SatZoo propose cette nouvelle allocation :

```
u_cnx_SC_SM
-u_cnx_RM_SC
-u_cnx_RM_SM
alloc_COM_SC
alloc_MON_SM
alloc_S1_SC
alloc_S2_SM
alloc_Equals_SM
alloc_Interrup_SM
used_SM
used_SC
 -used_RM
```

L'analyse des résultats se fait par l'interprétation du signe `-` devant chaque variable. En effet, lorsqu'une variable est précédée du signe `-`, cela signifie qu'elle n'est pas utilisée par le système. Par exemple la dernière linge : `used_SM used_SC -used_RM` nous indique que parmi toutes les ressources, `RM` n'est pas utilisée et donc pas nécessaire.

Cette allocation est intéressante car elle respecte bien l'ensemble des contraintes. Par contre, la différence qu'elle possède par rapport à celle étudiée dans le chapitre précédent est située au niveau de l'allocation de la donnée *dMON*. En effet cette donnée représentant la communication entre le *MON* et *Equals* qui était allouée au *Bus*. Or dans cette allocation, cette donnée n'a pas besoin de transiter par un bus puisque l'ensemble des tâches qui l'utilisent sont sur la même ressource de calcul (S_M). Généralement, on considère que les ressources de calcul possèdent un service de communication pour l'échange des données entre les tâches allouées à cette ressource.

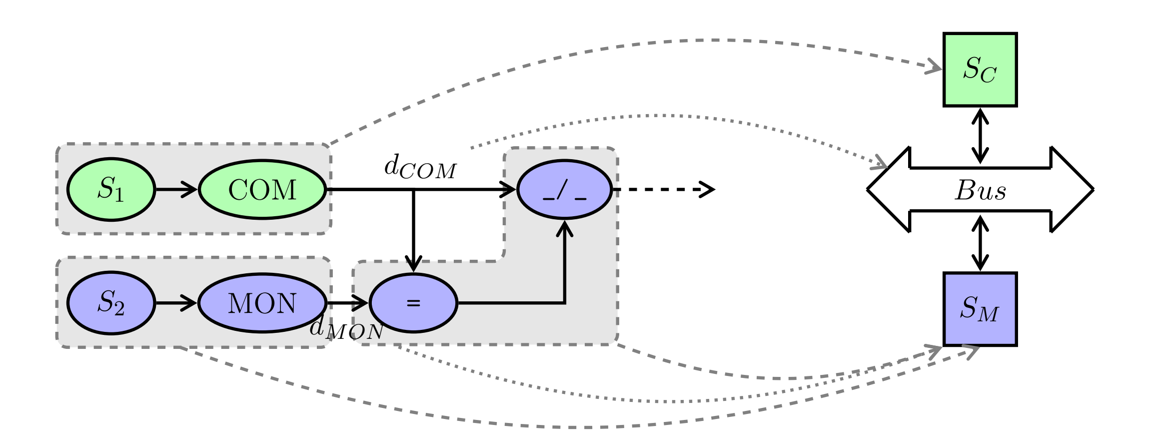


Fig. 4.14 – Une allocation minimale proposée par l'outil SatZoo

La figure précédente (4.14) ne montre pas la ressource de communication assurant la connexion entre R_M et S_M puisqu'elle n'est pas utilisée. De même,la figure ne montre pas la ressource inutile R_M.

4.7 Bilan

Dans ce chapitre nous avons montré comment en partant de la description des architectures fonctionnelles et matérielles et de directives d'allocation comme l'indépendance et la co-localisation, il était possible de produire automatiquement l'allocation des fonctions sur les ressources matérielles. Ceci passe par une étape de formalisation sous forme de contraintes linéaires entières puis par une étape de résolution de contraintes. Un schéma synthétisant cette démarche est proposé en figure 4.15.

Un outil supportant cette approche a été implémenté. Il permet d'assister le concepteur d'une architecture matérielle dans la recherche d'une allocation sûre.

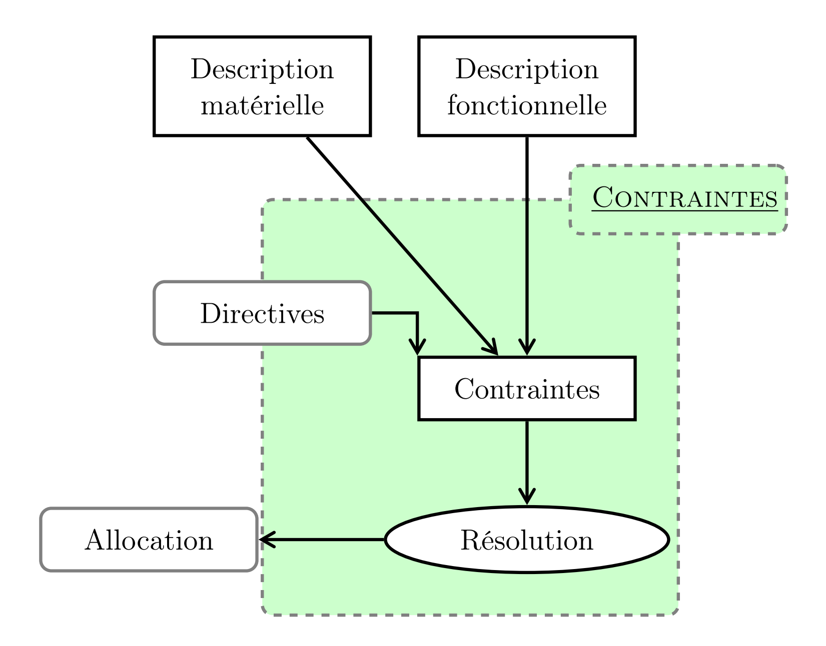


Fig. 4.15 – Démarche proposé dans ce chapitre

Chapitre 5

Méthode intégrée de recherche d'allocation

Dans cette partie, nous proposons une technique qui, partant du modèle d'architecture fonctionnelle et d'un modèle d'architecture matérielle, doit permettre de trouver une relation d'allocation. Cette démarche consiste dans un premier temps à extraire du modèle fonctionnel les hypothèses nécessaires pour valider nos exigences, puis à transformer ces hypothèses en contraintes pour orienter la recherche de la relation d'allocation. La résolution du système de contraintes permet de fournir aux concepteurs une première allocation ou lorsqu'il n'y a pas de solution, elle permet d'identifier les ressources pouvant poser problème.

Sommaire

5.1 Identifier les hypothèses d'indépendance

Le problème d'allocation consiste à trouver la relation *alloc* tout en garantissant certaines exigences de sûreté de fonctionnement. Nous proposons d'utiliser les techniques de validation des exigences vues dans la section 2.2.4 pour identifier les scénarios pertinents lors de la validation d'un système. L'analyse de ces scénarios nous permet d'identifier les fonctions qui doivent être allouées sur des ressources différentes pour ne pas invalider les exigences. Voyons en détail les deux techniques d'analyse de modèle AltaRica (génération de séquences et model-checking) qui vont permettre d'identifier les fonctions jouant un rôle important du point de vue de la sûreté de fonctionnement.

5.1.1 Identification par analyse de scénarios

Une première technique consiste à utiliser le générateur de séquence (cf. section 2.2.4) afin de produire un ensemble de scénarios minimaux menant à la situation redoutée (FC). Partant d'un ensemble de scénarios $\mathcal{S}c$ composé de p scénarios $\mathcal{S}c = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, C_i, \ldots, \mathcal{C}_p\}$ et connaissant le degré N de criticité de la situation redoutée, il faut identifier les fonctions qui ne doivent pas être allouées sur une même ressource.

Soit $\mathcal{C}$ un scénario dans $\mathcal{S}c$ composé de k événements : $\mathcal{C}_i = \{ev_1, ev_2, \ldots, ev_k\}$.

C représente donc une séquence minimale menant à la situation redoutée FC. Étant donné que cette séquence est produite par l'analyse du modèle de l'architecture fonctionnelle, tous les événements de cette séquence représentent la défaillance d'une fonction fx_i :

$$\mathcal{C}_i = \{fx_1.fail, fx_2.fail, \ldots, fx_k.fail\}$$

L'allocation des fonctions $(fx_1, fx_2 \ldots)$ sur les ressources $\mathcal{R}$ introduit des défaillances de mode commun (vu dans la section 1.2). Cela revient à remplacer la défaillance de la fonction fx_i par la défaillance de la ressource à laquelle elle est allouée (notée $alloc(fx_i)$). Nous obtenons ainsi une nouvelle séquence minimale $\mathcal{C}'$ incluant uniquement des défaillances des ressources de l'architecture matérielle.

$$\mathcal{C}' = \{alloc(fx_1).fail, alloc(fx_2).fail, \ldots, alloc(fx_k).fail\}$$

S'il existe des fonctions qui sont allouées sur une même ressource alors la défaillance de cette ressource va être représentée qu'une seule fois dans le scénario $\mathcal{C}'$. Dans ce cas, la taille de $\mathcal{C}'$ devient strictement inférieure à celle de $\mathcal{C}$, ce qui peut entraîner l'invalidation de l'exigence associée à FC

Suivant la classification de FC, le nombre minimal d'événements N composant les scénarios acceptables est donné dans par la table suivante :

FC	CAT	HAZ, MAJ	MIN
Sévérité (N)	3	2	1

Ainsi à l'aide de cette classification, nous pouvons examiner l'effet de l'allocation sur chaque scénario minimal de k événements ($C = \{fx_1.fail, fx_2.fail, \ldots, fx_k.fail\}$) :

- $k < N$: Ce scénario est inadmissible car il ne respecte pas l'objectif de safety.
- $k = N$: Ce scénario est admissible mais il faut absolument conserver sa taille lors de l'allocation. Il faut par conséquent que toutes les fonctions fx_i soient indépendantes :

$$\forall fx_1, fx_2 \in (fx_1^i, fx_2^i, \ldots, fx_k^i) : idp(fx_1, fx_2) = 1$$

- $k > N$: Ce scénario est admissible et on peut diminuer sa taille lors de l'allocation. Pour diminuer la taille du scénario (de façon acceptable) il suffit d'avoir l'indépendance de N fonctions parmi les k fonctions présentes dans le scénario.

Prenons à présent l'exemple du COM/MON (présenté dans le chapitre 3.1) afin d'illustrer comment identifier les indépendances entre les fonctions nécessaires. Partant de l'analyse du modèle fonctionnel du COM/MON, nous allons étudier les différents scénarios menant à une situation redoutée : FC exprimant que « la sortie du COM/MON est erronée ». Cette FC est classée MAJ donc les scénarios doivent contenir au moins 2 défaillances.

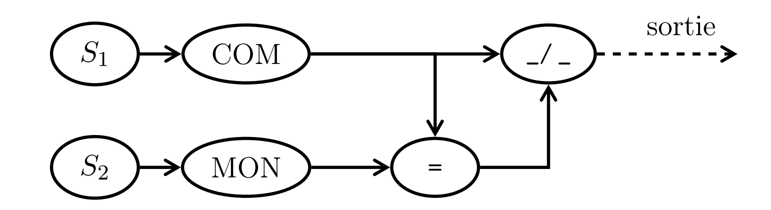


Fig. 5.1 – Modèle fonctionnel du *COM/MON*

Les différents scénarios menant à FC sont les suivants :

```
COM.fail_error & MON.fail_error;
COM.fail_error & S2.fail_error ;
 S1.fail_error & MON.fail_error ;
 S1.fail_error & S2.fail_error ;
```

Tous les scénarios possèdent 2 événements et comme l'indice de sévérité associé à l'exigence est aussi de 2, l'allocation ne peut introduire aucun partage de ressource entre ces fonctions. Les indépendances doivent donc s'appliquer sur toutes les fonctions correspondantes aux défaillances. Ainsi il faut définir les indépendantes suivantes :

- *idp*(COM , MON) = 1
- *idp*(COM , S2) = 1
- *idp*(S1 , MON) = 1
- *idp*(S1 , S2) = 1

Identification de contraintes de co-localisation

En complément de cette analyse sur l'indépendance, on remarque que S1 et COM (de même pour S2 et MON) possèdent les mêmes indépendances.

Si nous supposons que COM et S1 sont alloués sur la même ressource $Ress$, alors nous pouvons remplacer les défaillances de COM et de S1 par la défaillance commune $Ress_alloc$ ($Ress_alloc.fail = alloc(COM).fail = alloc(S1).fail$). Avec cette hypothèse, il n'apparaît pas de nouveaux scénarios d'ordre inférieur à 2 qui mènent à notre FC. Nous pouvons donc conclure que l'allocation de S1 et de COM sur une même ressource n'invalide pas l'exigence, car si on remplace la défaillance de S1 et celle COM par une défaillance commune représentant la perte de la dite ressource, alors il faut encore une défaillance pour atteindre la FC.

Cette remarque nous permet ainsi d'ajouter à notre système de nouvelles contraintes sur la co-localisation respective de COM et S1 et de MON et S2. Ces contraintes enrichissent notre système avec :

- *coloc*(COM,S1) = 1
- *coloc*(MON,S2) = 1

5.1.2 Identification des hypothèses d'indépendance par model-checking

La deuxième technique possible pour identifier les tâches qui doivent être indépendantes consiste à utiliser un model-checker afin de trouver toutes les combinaisons de pannes qui mènent à la situation redoutée. La technique mise en place utilise la traduction d'AltaRica dans le langage SMV et l'utilisation du model-checker associé.

Cette méthode est utile pour traiter des modèles dynamiques. Pour l'illustrer, nous transformons l'exemple de l'architecture COM/MON (cf. figure 5.1) :

Prenons l'exemple d'une situation redoutée correspondant à « la production d'une valeur incorrecte en sortie du COM/MON ». Cette valeur est produite par le composant *Interrup*, et est caractérisée par la variable `Out` de ce composant. Pour rendre l'exemple plus intéressant, nous ajoutons des défaillances à ce composant qui pour l'instant n'en possédait aucune. Son comportement actuel permet, en cas d'inégalité entre la valeur produite par COM et celle produite par MON, d'interrompre la liaison. Or, lorsque l'alarme est désactivée, la liaison est rétablie. Ce comportement n'est pas conforme au comportement réel d'une architecture COM/MON, car la liaison reste interrompue quoiqu'il arrive. Pour améliorer cela, nous proposons de modifier le comportement du noeud *Interrup* en ajoutant les deux événements suivants :

1. Un événement (`update`) est nécessaire pour ouvrir l'interrupteur en cas d'alarme.
2. À tout moment, l'interrupteur peut subir une défaillance (`fail_close`) le rendant dans la position bloquée fermée. Dans cette position, il est impossible d'interrompre le signal émis par COM.

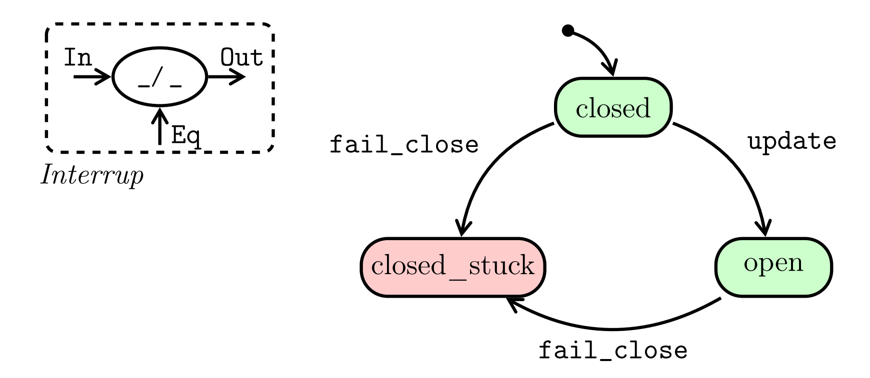


Fig. 5.2 – Nouveau comportement de *Interrup*

Pour obtenir un tel comportement, il suffit de remplacer le nœud AltaRica de l'interrupteur actuel par le nœud suivant :

```
node interrup
 state
   S:{open,closed,closed_stuck};
 flow
   Out:out:{correct,error,lost};
   In:in:{correct,error,lost};
   Eq:in:bool;
 event
  fail_close, update;
 trans
  Eq = false & S = closed |- update -> S := open ;
  S = closed or S = open |- fail_close -> S := closed_stuck ;
 assert
   Out = case {
     S = open : lost,
    else In
   };
 init
  S := closed;
edon
```

Après ces modifications du modèle il faut également modifier la formalisation de la situation redoutée. En effet, lorsque les voies de calcul COM et MON produisent des données différentes alors

l'interrupteur n'interrompt plus la fourniture de données instantanément. Ce n'est qu'après avoir joué la transition correspondant à l'événement `update` que l'interrupteur s'ouvre. Donc, pendant un instant il est possible qu'une donnée erronée soit propagée. La nouvelle situation redoutée que nous souhaitons considérer est la production de données erronées pendant au moins deux instants consécutifs.

Nous utilisons l'opérateur X (« Next ») de la Logique Temporelle Linéaire [Pnu77] (où « $X\phi$ » signifie que la propriété ϕ est vraie dans l'état suivant) pour modéliser cette nouvelle situation redoutée.

La formule : $S_{\mathcal{R}} \wedge XS_{\mathcal{R}}$ exprime que l'état courant correspond à une situation redoutée ($S_{\mathcal{R}}$ == (`Interrup.Out` = `error`)) et que l'état suivant correspond aussi à une situation redoutée. Par conséquent, cette formule nous permet d'identifier tous les états où le système reste dans une situation redoutée pendant 2 instants consécutifs.

Comme nous ne nous intéressons qu'aux combinaisons de pannes menant à la situation redoutée, une solution consiste à ajouter un compteur de pannes afin de dissocier les événements représentant les défaillances des autres événements (comme l'événement `update`). Pour cela, nous ajoutons au modèle SMV le composant $failure_count$ qui permet comme, son nom l'indique, de compter les défaillances.

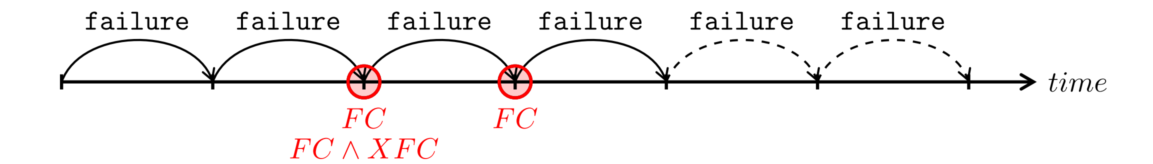
```
module failure_count(failure) {
/* failure is true whenever the current event is a failure */
 INPUT failure : boolean;
/* count counts the failure up to 3 */
 count : 0..3;
 init(count):=0;
/* the failure counter is incremented whenever
 a failure is the current  event and the maximal
 number of failures (3) as not been reached */
 next(count):= case{
  (failure) & ~(count=3) : count+1;
  default : count;};
}
```

Suivant le degré N de sévérité associé à la FC, nous cherchons à vérifier que tous les scénarios qui mènent à la FC sont d'un ordre supérieur à N. Par exemple, l'exigence qui nous intéresse est classée MAJ, ce qui correspond au niveau 2 de sévérité. Nous allons donc chercher à vérifier qu'il n'existe que des combinaisons d'au moins 2 défaillances élémentaires qui mènent à la situation redoutée considérée.

Cela peut se traduire en logique Temporelle Linéaire de cette façon :

$$F(S_{\mathcal{R}} \wedge XS_{\mathcal{R}}) \rightarrow F(failure_count \geq 2)$$

où « $F\phi$ » signifie que la propriété ϕ sera vraie dans un état futur (F pour « finally »).



Pour vérifier si une telle formule est valide pour le modèle considéré, nous utilisons le model-checker SMV de Cadence Labs [McM98]. Ce model-checker permet de prouver que le modèle de propagation satisfait bien cette formule.

Tant que la formule n'est pas vérifiée, le model-checker fournit un contre-exemple. L'analyse de ce contre-exemple permet d'extraire le scénario et donc d'identifier la séquence de défaillances qui invalide le modèle. Par conséquent, dans un premier temps, nous vérifions la première partie de la formule pour faire apparaître un premier scénario violant l'exigence. La première formule à vérifier est donc :

$$\neg F(S_{\mathcal{R}} \wedge XS_{\mathcal{R}})$$

Une fois que le premier scénario contre-exemple (cex_1) est identifié, nous cherchons à identifier de nouveaux scénarios invalidant eux aussi l'exigence. Pour cela, nous ajoutons à la formule précédente le scénario identifié et obtenons ainsi une nouvelle formule à vérifier :

$$\neg F(S_{\mathcal{R}} \wedge X S_{\mathcal{R}}) \vee cex_1$$

Si la formule n'est pas vérifiée, alors le model-checker fournit à nouveau un scénario contre-exemple (cex_2) différent de cex_1. De la même manière que précédemment, ce nouveau scénario est utilisé pour construire une nouvelle formule afin d'identifier (s'ils existent) de nouveaux scénarios contre-exemples.

$$\neg F(S_{\mathcal{R}} \wedge X S_{\mathcal{R}}) \vee cex_1 \vee cex_2$$

Nous réitérons cette technique jusqu'à obtenir la vérification de la formule :

$$\neg F(S_{\mathcal{R}} \wedge X S_{\mathcal{R}}) \vee cex_1 \vee \ldots \vee cex_n$$

Ainsi, de manière récursive, nous utilisons la génération de contre-exemples pour obtenir les différents scénarios invalidant l'exigence. Donc, lorsque l'ensemble des contre-exemples a été identifié, la formule suivante est vérifiée :

$$F(S_{\mathcal{R}} \wedge X S_{\mathcal{R}}) \rightarrow \bigvee_i cex_i$$

Maintenant que nous avons identifié l'ensemble des scénarios qui ne respectent pas l'exigence, nous allons extraire parmi ces scénarios les différentes fonctions à rendre indépendantes. Le fait d'imposer une indépendance entre fonctions va nous permettre de garantir qu'un scénario composé de 2 fonctions indépendantes est un scénario composé de 2 défaillances distinctes. Ainsi en rendant indépendantes les différentes fonctions intervenant dans les contre-exemples cex_i, nous garantissons qu'il faut au moins 2 défaillances pour atteindre la situation redoutée.

$$\bigvee_i cex_i \rightarrow F(failure_count \geq 2)$$

Appliquons à présent cette technique d'identification des indépendances sur l'exemple COM/MON.

Pour identifier le premier couple de défaillances à rendre indépendant, il faut vérifier la formule explicitant que le système ne peut pas rencontrer la situation redoutée étudiée :

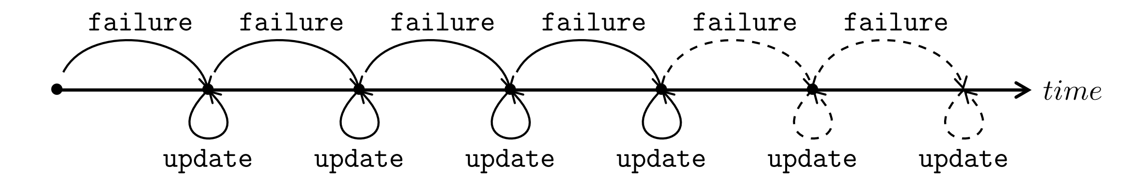
```
failure_cond :
  assert ~F((Interrup.Out = error) & X(Interrup.Out = error));
```

Pour forcer le système à se reconfigurer, nous ajoutons une hypothèse exprimant que l'événement `update` doit être tiré après chaque défaillance. Cette propriété est exprimée en logique temporelle :

```
Updates_follow_failure :
   assert G(failure -> X( event = update));

assume Updates_follow_failure;
```

Sous cette hypothèse, les traces d'éxécution qui nous intéressent deviennent de la forme :



Lors de la vérification de cette formule, le model-checker nous informe que le modèle ne satisfait pas cette propriété et nous le démontre en nous proposant le contre-exemple suivant :

```
using Updates_follow_failure prove failure_cond ;

-- CounterExample :
COM_fail_error ; update ; Interrup_fail_close
```

L'analyse du contre-exemple, permet d'identifier un premier couple de fonctions à rendre indépendantes à savoir : COM et Interrup. Pour identifier les autres relations d'indépendance nécessaires afin de valider la propriété considérée, il faut changer la propriété en prenant en compte le premier contre-exemple identifié. La propriété devient :

```
failure_cond :
assert  F(( Interrup.Out = error) & X( Interrup.Out = error)) ->
 (F(event = COM_fail_error) & F(event = Interrup_fail_close));
```

La vérification de cette formule nous propose un nouveau contre-exemple permettant l'identification d'une nouvelle indépendance entre fonctions. Pour l'exemple COM/MON, la vérification de la propriété est possible après identification des indépendances suivantes :

– idp(COM , Interrup) = 1
– idp(S1 , Interrup) = 1

Ainsi, de cette manière incrémentale, nous construisons une formule qui peut être généralisée par la suivante :

$$F(S_{\mathcal{R}} \wedge XS_{\mathcal{R}}) \rightarrow \bigvee_{ev_i, ev_j \in idp} (F(ev_i) \wedge F(ev_j))$$

où (ev_i, ev_j) correspond à une paire de défaillances élémentaires.

Afin de prouver l'exigence de sûreté, nous devons nous assurer que toutes les défaillances regroupées en paires sont indépendantes. Nous dérivons les hypothèses d'indépendance de la forme :

$$\forall i, j | idp(ev_i, ev_j) : F(ev_i) \wedge F(ev_j) \rightarrow F(failure_count \geq 2)$$

où ($event_i$, $event_j$) est une paire de défaillances apparaissant dans la partie droite de la formule. En utilisant ces hypothèses ainsi qu'une hypothèse supplémentaire spécifiant que seules les tâches peuvent subir une défaillance, nous pouvons prouver que l'exigence de sûreté est respectée par le modèle COM/MON.

5.2 De l'hypothèse d'indépendance vers la contrainte de ségrégation

La validation des exigences de sûreté de fonctionnement sur un modèle de système n'est possible que sous certaines hypothèses. Comme nous l'avons vu précédemment, un ensemble d'hypothèses sur l'indépendance de fonctions est nécessaire pour vérifier ces exigences. Les hypothèses d'indépendance que nous avons identifiées dans la section précédente nous indiquent, par exemple, que la fonction f_1 ne doit pas être allouée sur la même ressource que la fonction f_2, si l'on souhaite vérifier certaines propriétés. Nous proposons donc de traduire ces hypothèses en contraintes (directives) de ségrégation afin de n'obtenir (lors de la résolution du système de contraintes) que des allocations préservant les exigences de sûreté. La démarche ainsi proposée est schématisée par la figure 5.4. Elle reprend les démarches établies dans les chapitres précédents (cf. 3.23 et 4.15) en y ajoutant une méthode d'intégration. On peut lire le schéma de la façon suivante :

– Partant d'une description des architectures fonctionnelles et matérielles, les modèles AltaRica de ces architectures sont réalisés.
– L'analyse du modèle fonctionnel fournit des résultats dont on extrait les directives d'indépendances.
– Les directives s'ajoutent aux contraintes dérivées de la description des architectures fonctionnelles et matérielles. La résolution de ces contraintes produit une allocation des fonctions sur l'architecture matérielle.

Les solutions proposées par le « *solveur* » de contraintes préservent les exigences de sûreté. Ces nouvelles contraintes permettent donc de réduire l'espace des solutions possibles à un espace de solutions respectant les exigences de sûreté (cf. figure 5.3).

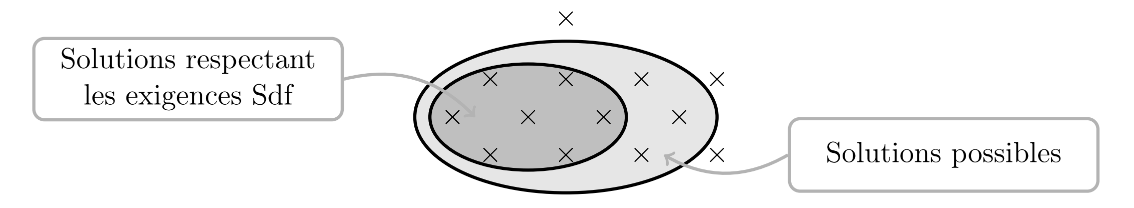


Fig. 5.3 – Les solutions n'invalidant pas les exigences de sûreté

– La nouvelle allocation obtenue par résolution du système de contraintes peut être aisément ajoutée au modèle AltaRica pour effectuer de nouvelles analyses. En effet, comme vu dans le chapitre 3, la relation d'allocation peut-être modélisée en AltaRica par l'ajout d'un vecteur de synchronisation. Ce vecteur exprime une défaillance commune représentant la défaillance de tous les composants liés par la relation d'allocation.
Ce nouveau modèle AltaRica est dit « global » car il représente dans un même modèle : l'architecture fonctionnelle et l'architecture matérielle. À partir du modèle global, il est possible d'effectuer de nouvelles analyses. Par exemple, il est possible de valider des exigences quantitatives ou des exigences qualitatives en prenant en compte l'effet des défaillances des ressources matérielles (autorisé par l'allocation).

Fig. 5.4 – Technique d'obtention d'une allocation

Les exigences quantitatives permettent de définir une probabilité de défaillance dangereuse pour une situation redoutée. Cette probabilité est calculée en fonction de la probabilité associée à chacune des défaillances des ressources de l'architecture matérielle et en fonction de l'allocation. En effet,

suivant l'allocation choisie, les combinaisons de défaillances matérielles menant à la situation redoutée varient et par conséquent, la probabilité d'apparition de la situation redoutée évolue suivant le choix de l'allocation analysée. Ces exigences ne peuvent donc être vérifiées qu'une fois le choix de l'allocation établi.

Une deuxième analyse rendue possible par l'ajout de l'allocation dans le modèle AltaRica est la simulation du système dans une configuration architecturale réelle. Cette simulation permet de visualiser l'effet des défaillances matérielles sur l'architecture fonctionnelle étant donné une allocation.

De plus il est possible de vérifier que les exigences de sûreté de fonctionnement satisfaites par l'architecture fonctionnelle sont effectivement préservées après allocation sur l'architecture matérielle. A priori cette vérification est inutile puisque les allocations fournies par le solveur préservent les exigences de l'architecture fonctionnelle. Malgré ceci, cette vérification peut être intéressante dans le contexte de certification où l'on apprécie de disposer de plusieurs moyens indépendants pour obtenir certains éléments de confiance. Cette vérification pourrait permettre de détecter d'éventuels problèmes lors de l'identification des contraintes de ségrégation ou lors de la résolution des contraintes.

5.3 Adaptation de l'architecture matérielle

Nous présentons dans cette partie une amélioration de notre méthode de résolution de contrainte.

Lorsqu'une architecture existante ne satisfait pas les exigences qui lui sont associées, il est souhaitable de pouvoir proposer aux architectes une justification de la non satisfaction des exigences, et de proposer une architecture minimisant les modifications à apporter pour rendre l'architecture validante. Cela permet d'éviter de remettre en cause l'ensemble d'une architecture donnée, ce qui est favorable à la réutilisation et l'adaptation d'architectures d'un avion à l'autre.

Lorsqu'il n'existe pas d'allocation possible pour une architecture donnée (le système d'équations linéaires décrivant les contraintes n'a pas de solution), nous avons élaboré une méthode[1] permettant d'identifier dans l'architecture proposée, la (ou les) ressource(s) à modifier pour obtenir une bonne allocation.

Deux configurations peuvent rendre impossible la définition d'une allocation conforme aux contraintes :

1. un système de contraintes impossible à résoudre
2. une architecture matérielle incompatible avec les contraintes.

Pour se prémunir du premier cas, il est nécessaire de vérifier la cohérence de l'ensemble des contraintes. Dans le second cas, il est nécessaire de modifier l'architecture matérielle pour la rendre compatible des exigences du système.

Nous nous intéressons ici à ce second cas, dans lequel l'architecture matérielle proposée ne possède pas suffisamment d'équipements pour permettre une allocation correcte.

Nous souhaitons donc pouvoir identifier dans l'architecture les types de ressources dont le nombre est insuffisant. Pour faciliter cette identification, nous ajoutons à notre système de nouvelles variables pour repérer les *conflits* dans l'architecture, que nous définissons ci-après.

Un *conflit* sur une ressource intervient lorsqu'au moins deux fonctions indépendantes sont alloués à une même ressource.

[1] Méthode spécifiée mais non implémentée

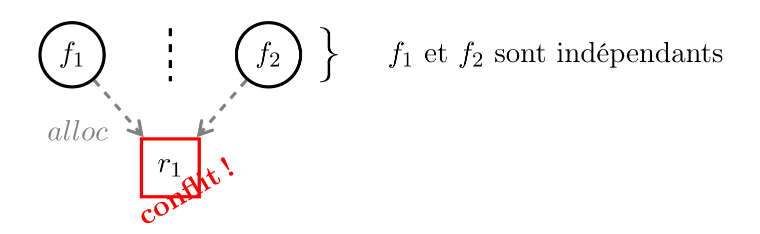


Fig. 5.5 – Notion de conflit sur une ressource

Cette nouvelle variable permet d'établir, pour chaque ressource $r \in \mathcal{R}$, si plusieurs fonctions indépendantes lui sont allouées. Ainsi, *conflit(r)* est définie de la manière suivante :

$$conflit: \quad \mathcal{R} \rightarrow \{0, 1\}$$
$$conflit(r) = 1 : \text{au moins 2 fonctions indépendantes sont allouées à } r.$$

Finalement, cette variable permet d'identifier les ressources posant des difficultés pour l'allocation. Elle permet donc d'aider un architecte à cibler la partie de l'architecture matérielle à modifier. La problématique de l'identification des conflits est présentée dans l'article [PBR03].

Minimiser le nombre de conflits Un critère d'optimisation permet de réduire le nombre de solutions proposées pour ne calculer que la meilleure relativement à ce critère. Le critère d'optimisation utilisé dans nos travaux (nombre de connexions) peut être remplacé par le nombre de conflits. En minimisant le nombre de conflits, la solution proposée correspond à la solution possédant le moins de conflits de ressources. Ainsi, la solution obtenue représente l'architecture la plus simple à modifier (en terme d'ajout de ressources) et les modifications à effectuer sont guidées par les conflits de ressources associés.

Ce nouveau critère s'écrit donc :

$$\forall r \in \mathcal{R} : c_min = \sum conflit(r)$$

Approche incrémentale Les variables de conflits permettent donc de trouver une allocation malgré le non-respect de certaines contraintes. L'ajout de ces variables et le changement du critère d'optimisation permettent d'identifier rapidement les ressources ne respectant pas les contraintes.

Une méthode a été développée afin de combiner la recherche d'allocation et lorsque cette recherche ne fournit pas de résultat, l'identification des ressources posant problème. Cette technique permet d'automatiser le processus d'obtention d'une première allocation possible. Elle consiste à rechercher progressivement l'allocation en modifiant (lorsque cela s'avère nécessaire) l'architecture proposée.

La démarche proposée peut être synthétisée par le diagramme de la figure 5.6. Partant d'une description de l'architecture et d'un ensemble de directives, il est possible de construire le système de contraintes correspondant. Lorsqu'il n'existe pas d'allocation, il suffit d'analyser les conflits pour identifier les ressources posant problème dans l'architecture. Une solution de remplacement de la ressource posant problème par deux nouvelles ressources permet de modifier l'architecture et de relancer les analyses à la recherche d'une allocation possible.

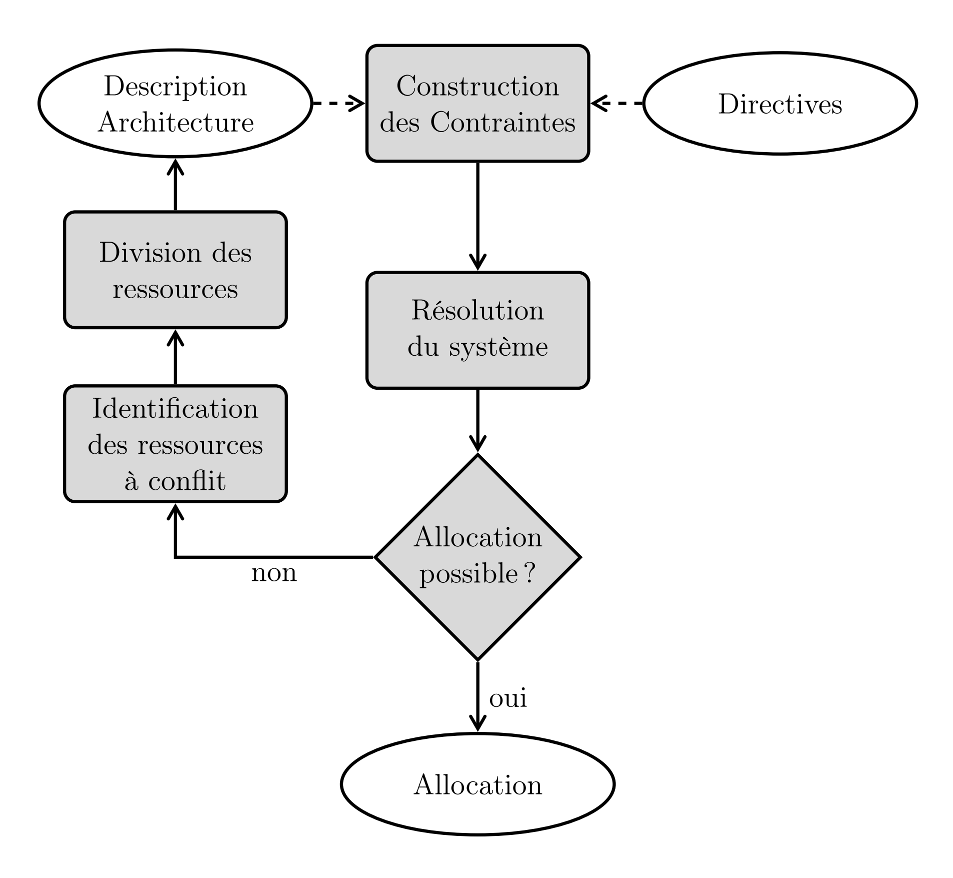


Fig. 5.6 – Démarche pour la recherche d'allocation incrémentale

Comme décrit précédemment, un conflit existe sur une ressource si au moins deux fonctions indépendantes y sont allouées et par conséquent cette ressource viole systématiquement l'exigence sur l'indépendance de ces fonctions. La première modification de l'architecture matérielle nécessaire pour pallier à la violation de cette exigence consiste à remplacer ladite ressource par un sous ensemble de ressources possédant les mêmes caractéristiques. Cette technique appelée *division* permet de modifier simplement une architecture existante par l'ajout de ressource.

Lorsqu'une ressource est divisée, nous considérons que toutes ses connexions doivent être reportées sur les différentes ressources produites.

Une fois toutes les divisions nécessaires effectuées, il suffit de relancer les analyses pour obtenir une allocation (un exemple est proposé dans la figure 5.7).

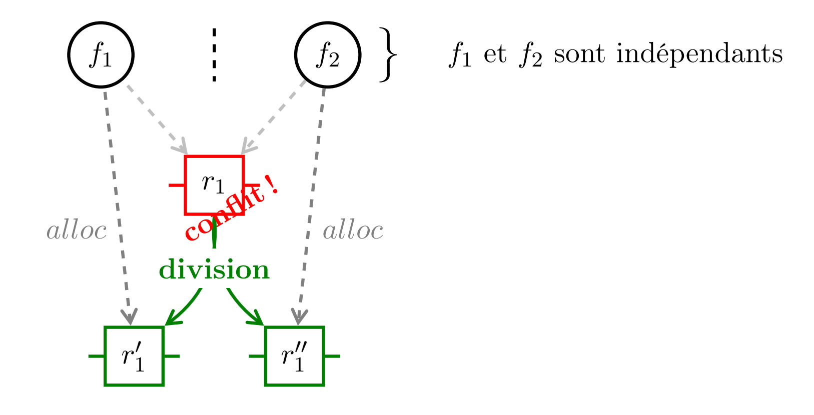


Fig. 5.7 – Technique de *division* d'une ressource en conflit

5.4 Bilan

Dans ce chapitre, nous proposons une méthode qui permettant de produire automatiquement des directives d'indépendance qui sont nécessaire pour respecter les exigences de système.

Cette méthode est applicable dans le cas des modèles statiques par analyse des scénarios générés par les outils. Elle est aussi appliquée dans le cas des modèles dynamiques en se fondant sur la technique de *model-checking* .

Cette méthode permet de relier l'approche modélisation du chapitre 3 avec l'approche génération de contraintes proposée dans le chapitre 4.1.

Finalement, nous avons évoqué une extension possible e l'approche qui cherche à proposer des solutions alors que le système de contraintes initial ne permet pas de produire une allocation.

Troisième partie

Mise en application de la méthode proposée

Chapitre 6

Le Système de Suivi de Terrain (SdT) d'un avion de chasse

L'approche développée a été validée sur un système qui contribue au pilotage automatique d'un avion militaire. Il s'agit précisément du système de suivi de terrain (noté SdT par la suite) lié au projet Carlit[a] élaboré par l'ONÉRA. Après une brève présentation de ce système et des différents éléments le constituant, nous présentons nos choix de modélisation ainsi que les différentes étapes permettant la vérification des exigences que ce système doit garantir.

[a] Cœur Avionique ReconfigurabLe InTégré

Sommaire

A la différence de l'exemple traité dans le chapitre précèdent, il ne s'agit pas d'un système de type informatique. Malgré cela nous montrons que l'approche de modélisation et de vérification de l'allocation est applicable.

6.1 Description du système de Suivi de Terrain

Le principe du système de suivi de terrain *SdT* [adl00] est de permettre d'élaborer un pilotage automatique ou manuel dans le plan vertical (selon le choix du pilote influencé par les conditions météorologiques) à partir des informations de terrain fournies par le Radar et le pilote (préalablement enregistrées) et des informations de navigation (fournies par les différents capteurs au cours du vol). La propriété principale de ce pilotage est de permettre le franchissement des différents obstacles du terrain à très basse altitude (minimum 60m) en maintenant un haut niveau de sécurité. Ce mode de vol à basse altitude permet de rendre l'avion le moins détectable possible.

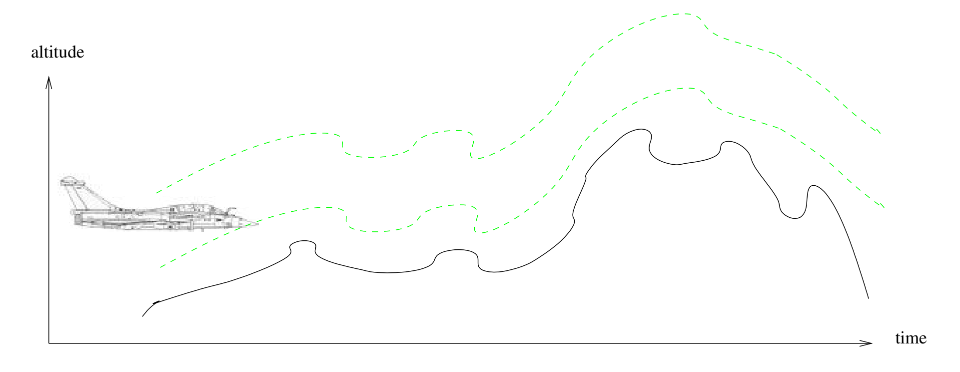


Fig. 6.1 – Illustration du Suivi de Terrain

Ce *SdT* peut provoquer, en cas de situation de vol critique, un ordre de dégagement permettant une montée très rapide de l'avion par rapport à sa dernière position fournie par les capteurs. Ainsi, il est primordial d'assurer que les informations utilisées pour ce dégagement déterminent correctement la position de l'avion, afin de donner une bonne orientation à ce dégagement.

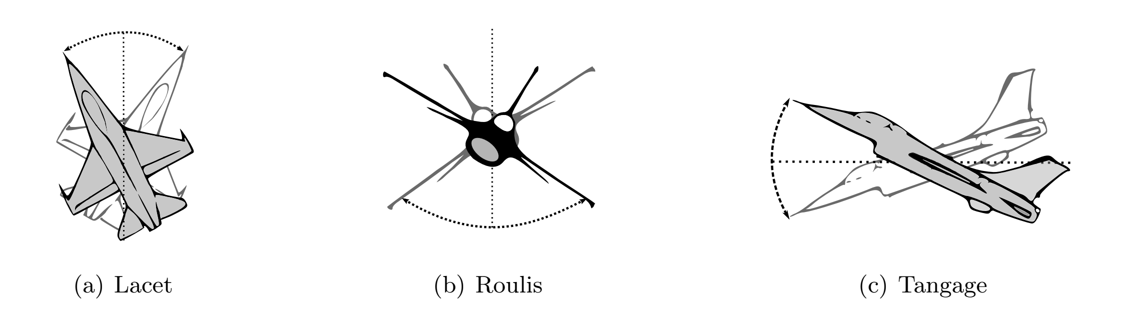
Selon l'approche Carlit, le *SdT* est défini par un ensemble de chaînes fonctionnelles associées à un certain mode. Ces chaînes fonctionnelles sont constituées par des fonctions reliées entre elles par des échanges d'informations.

Chaque fonction intervenant dans les chaînes du *SdT* est préalablement spécifiée indépendamment des fonctions avec lesquelles elle interagit au sein de la chaîne. Chaque fonction est ainsi décrite par sa tâche et par les types de données qu'elle utilise. La fonction connaît ses types d'entrée mais ne sait de quelles fonctions (ou de quels capteurs) ils proviennent.

Dans l'étude de cas, nous nous sommes tout particulièrement intéressés au mode fonctionnel automatique du *SdT* (*S*dT.Automatique) : ce mode correspond à un pilotage automatique de l'avion ; les ordres de pilotage sont directement adressés aux commandes de vol (CDVE).

Dans le mode « Automatique », le système *SdT* calcule en permanence des ordres de pilotage en « profondeur » qu'il envoie directement aux commandes de vol permettant ainsi le contrôle de l'avion dans le plan vertical. En parallèle, le système *SdT* surveille en permanence la cohérence et le bon fonctionnement des ressources et des tâches, et produit en cas d'anomalie une alarme de dégagement. Cette alarme est suivie d'une manoeuvre de dégagement qui permet de positionner rapidement l'avion à une altitude suffisante afin d'éviter le relief et permettre ainsi au pilote la reprise du contrôle de l'appareil. Cette manoeuvre est basée sur le calcul d'un angle de tangage (*Pitch*) (c.f. figure 6.2) suffisant afin d'éviter tout relief. Avant que l'alarme ne déclenche ce « dégagement », le système *SdT* calcule un angle de roulis à appliquer afin de stabiliser l'avion à l'horizontale (c.f. angle *Roll* de la figure 6.2). En effet, il est indispensable pour le bon déroulement de ce dégagement que l'avion soit « horizontal » sinon il pourrait être inopérant ou même conduire à la perte de l'appareil.

Il existe une autre mode dit manuel où les ordres de pilotage sont envoyées sur les écrans de visualisation et c'est au pilote de suivre ou non ces ordres. Finalement il existe un mode dans lequel toutes les protections (alarme liée à une altitude dangereuse et commande de dégagement automatique) sont inhibées.



(a) Lacet

(b) Roulis

(c) Tangage

Fig. 6.2 – Mouvements possibles de l'avion

6.2 Description du système

6.2.1 Architecture fonctionnelle

Le rôle du système considéré est de maintenir une altitude minimale en fonction d'un plan de vol décidé à l'avance par le pilote. Cette altitude varie suivant le terrain survolé par l'appareil.

L'ensemble des fonctions utilisées par le système est schématisé par la figure 6.3.

Afin de s'adapter constamment au relief du terrain, le système est doté d'un *Radar* (que nous considérons par la suite comme un capteur fournissant en permanence un flux de données) permettant de lui fournir des images du terrain (*Image_ Terrain*). Les données produites par ce capteur sont comparées aux *Données_ terrain* (données préalablement enregistrées dans une base de données propre à la mission nommée *BD_ Terrain*) pour vérifier que l'appareil suit la trajectoire voulue.

Étant dans le mode « Automatique », le système contrôle aussi la vitesse de l'appareil. Le système reçoit plusieurs données des équipements de *Navigation* pour connaître sa position ainsi que ses différentes vitesses (vitesse dans le plan vertical *VS* et horizontal *VZ*) afin de conserver la vitesse la plus élevée. Ce composant *Navigation* constitue un des éléments les plus critiques, car il fournit les différentes données nécessaires pour calculer les ordres de pilotage. Ce composant renvoie aussi des données permettant d'analyser la position de l'avion par rapport au plan horizontal (*CCS*). Cette position permet le calcul de l'angle de tangage à appliquer afin de franchir les différents obstacles dus au terrain.

Les autres données nécessaires au système sont fournies par le Radio Altimètre (*RA*) et le *P_ SdT*.

- Le radio altimètre est une sonde permettant de connaître à tout moment la hauteur de la structure par rapport au sol (*HRS*). Cette donnée permet donc de contrôler l'altitude réelle de l'appareil et est importante lors du calcul du dégagement.
- Le *P_ SdT* est un appareil de saisie permettant au pilote de renseigner différents paramètres propres à une mission (*HS*,*HO*...)

Dans ce mode « Automatique », le système *SdT* calcule en permanence des ordres de pilotage en profondeur (*DOLONGI*) à l'aide de différentes fonctions d'élaboration (*Élaboration ordre terrain*, *Élaboration ordre altitude*, *Élaboration ordre en profondeur*, etc.). Ces ordres de pilotage sont ensuite directement envoyés aux commandes de vol (*CDVE*).

En parallèle, le système *SdT* surveille en permanence la cohérence et le bon fonctionnement des ressources et des tâches, et produit en cas d'anomalie une *alarme de dégagement* calculée par une

fonction d'*Élaboration d'alarme de dégagement* et renvoyée sur les différents contrôles du pilote (Haut-parleur : *HP_SdT*, Écran de contrôle : *Display_SdT*).

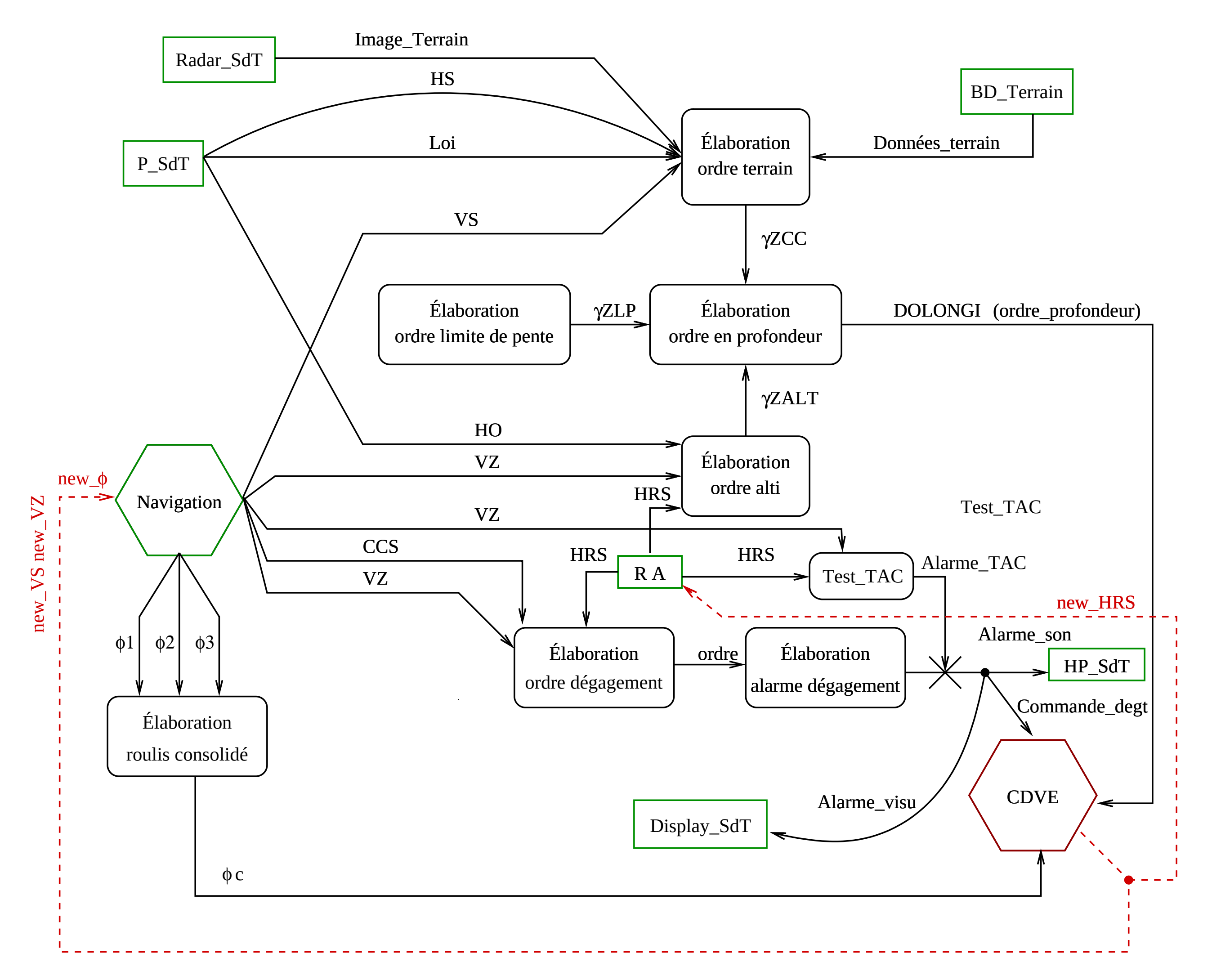


FIG. 6.3 – Architecture fonctionnelle du Suivi de Terrain

6.2.2 Architecture matérielle

Le système *SdT* est un système distribué fonctionnant sur une architecture de type *IMA*[1]. Dans ce type d'architecture, les ressources de calcul et de communication sont partagées par les différentes fonctions du système. L'architecture du système est représentée par la figure 6.4. Cette figure illustre comment les différentes fonctions du système partagent le même réseau de communication (*IMA*).

[1] Integrated Modular Architecture

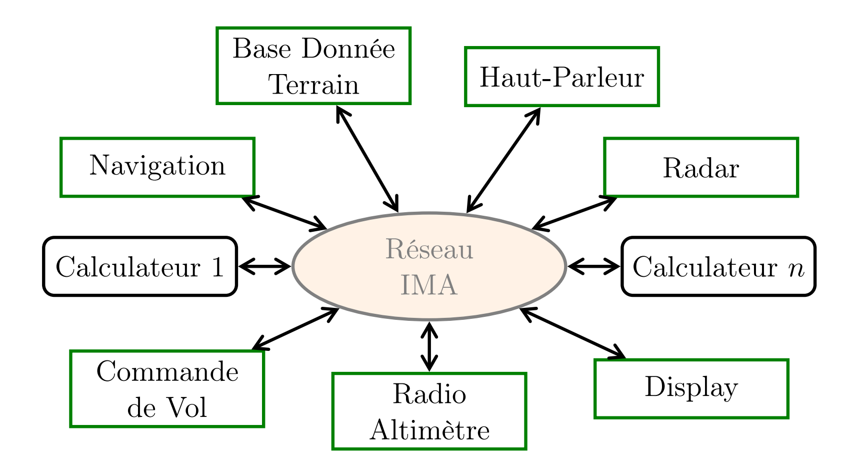


Fig. 6.4 – Architecture matérielle du Suivi de Terrain

L'utilisation d'une telle architecture permet le partage de certaines ressources par plusieurs applications et ainsi minimise le nombre de ressources nécessaire au bon fonctionnement de l'avion. Les intérêts d'une telle architecture sont discutés dans [BLED99]. Les fonctions du système se partagent les différents calculateurs connectés à ce réseau, ce qui va engendrer de difficultés nouvelles lors de la conception de l'architecture. En effet, il n'est pas toujours aisé de trouver les choix d'allocation de fonctions sur les calculateurs qui vont garantir le bon déroulement de la mission en cas de défaillance d'un ou plusieurs calculateurs. La technologie *IMA* et le partage des ressources qu'elle introduit engendrent de nouvelles défaillances de mode commun à prendre en compte lors des analyses.

6.3 Modélisation du système

6.3.1 Modèle fonctionnel

Le système *SdT* est composé de capteurs qui produisent un ensemble de données. Ces données sont ensuite envoyées à des fonctions afin de calculer un ordre nécessaire aux commandes de vol. Ces données représentent des valeurs réelles calculées par des fonctions ou fournies par les capteurs. Par exemple, le radioaltimètre permet de déterminer la position de l'avion par rapport au sol. Cette donnée représentant l'altitude doit être la plus précise possible afin d'effectuer les bons repérages du terrain pour permettre (en cas d'activation du pilotage automatique) de maintenir l'avion à l'altitude la plus basse possible. Étant donné que l'analyse que nous souhaitons effectuer sur ce système est orientée sur le comportement global du système en présence de défaillances, nous ne considérons pas la valeur réelle de cette donnée d'altitude mais uniquement une valeur abstraite dénotant l'occurrence d'une défaillance en amont dans la chaîne de traitement de cette donnée.

Une première abstraction nécessaire pour la modélisation de ce système concerne les données échangées. Pour nos analyses, nous considérons trois types de valeur (dénotés par la figure 6.3.1) :

- `correct` : Cette valeur permet de représenter l'ensemble des données n'ayant pas subi de défaillance et dont la valeur réelle appartient à l'intervalle de valeurs pour d'un fonctionnement correct.

- `erroneous` : En cas d'occurrence d'une défaillance dans les fonctions nécessaires à la production d'une donnée, nous considérons que la donnée est influencée par cette défaillance et nous supposons par exemple que le comportement erroné de la fonction défaillante a pour conséquence que la valeur réelle représentée par cette donnée n'est plus dans l'intervalle des données acceptables. Une telle représentation des données permet de visualiser l'impact d'une donnée erronée sur les fonctions l'utilisant. Étant donné que le *SdT* est constitué d'un ensemble de fonctions s'échangeant des données, il est important d'observer le comportement global du système en cas

de présence d'une donnée n'ayant pas une valeur attendue.

– `lost` : Comme défini précédemment dans le chapitre 3, il existe une défaillance sur les fonctions qui les empêche de produire leur résultat. Cette valeur permet d'observer l'influence d'une absence de donnée nécessaire pour le système.

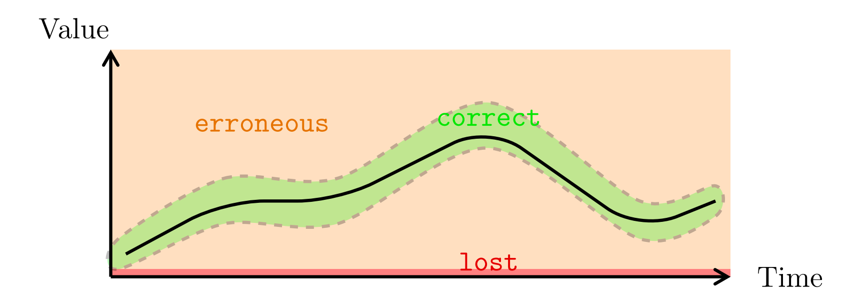


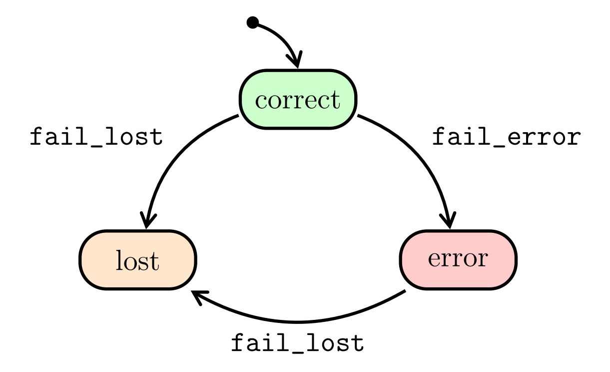
Fig. 6.5 – Abstraction de la valeur d'une donnée

Dans la suite de ce chapitre, lorsque ce type sera utilisé dans les composants AltaRica il sera représenté par le domaine suivant :

```
domain FailureType = {correct, erroneous, lost};
```

Une fois que l'abstraction des données échangées est effectuée, il faut identifier le niveau d'abstraction nécessaire pour la modélisation des fonctions et autres composants du système. Le niveau d'abstraction choisi pour cette analyse est celui défini dans le chapitre 3. Les différents composants du système peuvent ainsi subir plusieurs défaillances : `fail_error` et `fail_lost`. Ces défaillances permettent d'illustrer le comportement erroné d'une fonction ainsi que la perte de ladite fonction (dans notre cas, nous considérons qu'une perte de fonction correspond à la non-production de la donnée associée).

L'automate correspondant aux différents changements d'état d'une fonction (section 3.2 en page 33) est rappelé dans la figure suivante :



Ce niveau d'abstraction doit être appliqué à l'ensemble des éléments composant le système *SdT*. Pour cela, il suffit de créer pour chaque composant du système, son correspondant en AltaRica. L'analyse des différents composants du système permet d'identifier plusieurs familles de composants :

- Les capteurs : composants produisant des données. Nous utilisons deux types de capteurs :
  - Capteur source. Ce capteur n'a pas d'entrée et a une sortie unique. Ce type de comportement est associé aux fonctions Radar, TFTAPanel, Roll1 et Roll2. le code AltaRica associé à ce composant a été décrit dans la section 2.2.3.
  - Capteur bouclé. Ce capteur a une entrée qui est branchée sur la sortie du système SdT. Ce type de composant est associé aux fonctions Navigation, RadioAltimeter. En l'absence de défaillance, ce composant propage sur son port de sortie ce qui est reçu sur le port d'entrée.

– Fonction de calcul : ce composant permet à partir de plusieurs données d'en produire une nouvelle. Ce composant est associé aux fonctions `VerAccComp`, `ClAlarmComp` et `CommandeVol`.
– Fonction de consolidation : ce composant permet de calculer une donnée consolidée à partir de deux données provenant de sources différentes. Ce composant est associé `ConsRollComp`.
– Flux de données : ce composant permet de transporter des données échangées par des fonctions. ce composant est associé à toutes les flux de données du système SdT.

Ce regroupement de composants en différentes familles permet de factoriser les composants AltaRica à utiliser et ainsi construire une bibliothèque d'éléments à utiliser. Sachant que les fonctions n'utilisent pas le même nombre de données pour le calcul de leur donnée, elles sont regroupées selon le nombre de données qu'elles utilisent. Par exemple, la fonction de calcul d'une Accélération verticale (`VerAccComp`) requiert 3 données (`TerrainInfo`, `SHeight` et `Speed`) pour calculer l'ordre de *Pitch* à renvoyer aux commandes de Vol. Par contre, la fonction calculant l'alarme de dégagement n'utilise que les données `Alt` et `Vspeed`. Par conséquent, même si ces fonctions possèdent le même niveau d'abstraction du point de vue de leur comportement fonctionnel, elles sont distinguées par des interfaces différentes. Il est donc nécessaire de leur associer des composants différents.

Le code AltaRica pour une fonction de calcul n'ayant que 2 données à traiter est le suivant :

```
node Function_2in
 flow
  In1 : FailureType : in ;
  In2 : FailureType : in ;
  Out : FailureType : out ;
 state
  Status : FailureType ;
 event
  fail_error, fail_lost ;
 init
   Status := correct ;
 trans
  Status = correct |- fail_error -> Status := erroneous;
  not(Status = lost) |- fail_lost -> Status := lost;
 assert
  Out = case {
    Status = correct and (In1 = correct and In2 = correct) : correct,
    Status = correct and (In1 != correct or In2 != correct) : erroneous,
    Status = erroneous : erroneous,
    Status = lost : lost,
    else erroneous };
edon
```

La fonction de calcul est soumise à deux types de défaillances : l'erreur est modélisée par l'évènement `fail_error`, la perte est modélisée par l'évènement `fail_lost`. L'assertion décrit son comportement en présence des défaillances :
– si elle n'a subi aucune défaillance et que toutes ses entrées sont correctes elle fournit un résultat correct
– sinon, si elle n'a subi aucune défaillance et qu'une de ses entrées n'est pas correcte elle fournit un résultat incorrect
– sinon, si elle est en erreur elle produit un résultat erroné,
– et sinon, si elle est perdue, elle ne produit pas de résultat

Le comportement de la fonction de consolidation est donné par le code AltaRica suivant :

```
node Function_Consolidation
 flow
  In1 : FailureType : in ;
  In2 : FailureType : in ;
```

```
  Out : FailureType : out ;
 state
  Status : FailureType ;
 assert
  Out = case {
     (In1 = erroneous and In2 = erroneous) : erroneous,
     (In1 = correct and  In2 = correct)    : correct,
     else                                    lost));
edon
```

Nous avons supposé pour cette étude que la fonction de consolidation n'était pas soumise à des défaillances. Le comportement de cette fonction est le suivant :

- Si les deux entrées sont erronées alors la fonction produit une donnée erronée,
- Si les deux entrées sont correctes alors la fonction produit une donnée correcte,
- dans tous les autres cas, la fonction est capable de détecter qu'au moins une des deux entrées est perdue ou incorrecte dans ce cas la fonction ne produit pas de résultat.

Une fois que la correspondance entre chaque élément du système avec un composant AltaRica est faite, il ne reste plus qu'à interconnecter les composants AltaRica entre-eux tout en respectant les connexions déjà établies entre les éléments du système. Ces interconnexions vont ainsi permettre d'obtenir un modèle fonctionnel décrivant le comportement global du système ayant la particularité de pouvoir être testé en présence de défaillances.

Une représentation graphique du Système *SdT* en AltaRica est proposée par l'utilisation de l'outil *Cecilia OCAS*. Elle est représentée par la figure suivante :

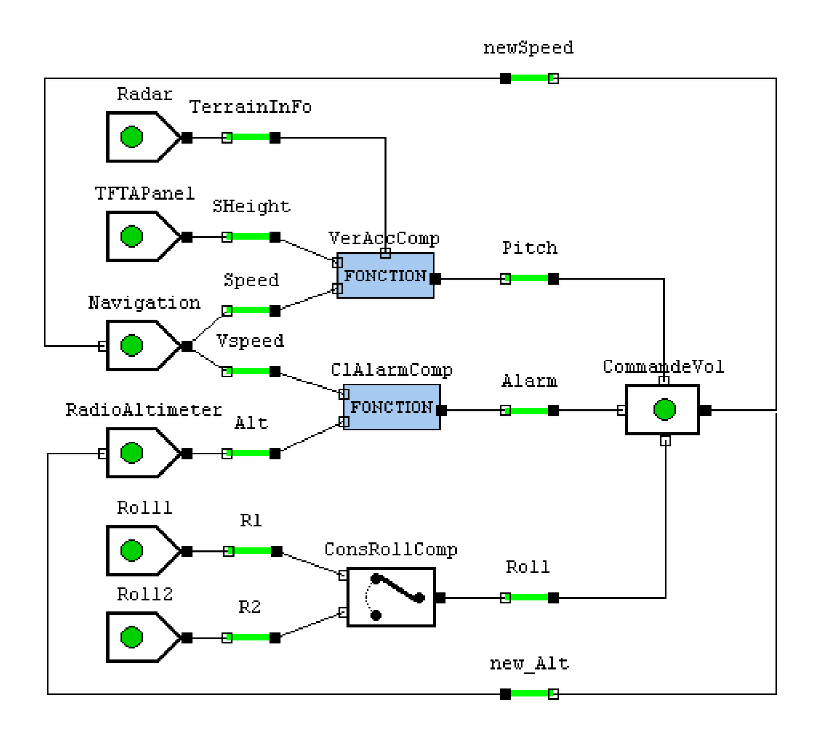


Fig. 6.6 – Modèle AltaRica du SdT

Une des particularités du modèle global du *SdT* est qu'il possède une boucle d'interdépendance entre valeurs, empêchant l'initialisation des valeurs. Cette boucle est facilement identifiable sur le modèle car la sortie du composant `CommandeVol` est branchée sur les entrées des capteurs `Navigation` et `RadioAltimeter`. Ceci a pour effet de créer dans le modèle des définitions circulaires qu'il faut éviter pour créer un modèle conforme à la syntaxe *AltaRica Data Flow*. Pour cela, nous ajoutons un délai dans le comportement associé à la fonction `CommandeVol`. La sortie `Out` n'est pas modifiée instantanément

en fonction des nouvelles valeurs de ses entrées. Elle n'est modifiée qu'après le tirage d'un événement interne `update` qui modifie une variable d'état `pre_Out`.

```
node Function_3in
 flow
  In1 : FailureType : in ;
  In2 : FailureType : in ;
  In3 : FailureType : in ;
  Out : FailureType : out ;
 state
  pre_Out : FailureType ;
 event
  update ;
 init
   pre_Out := correct ;
 trans
  (In1 != correct or In3 != correct)
  and (not (pre_Out = erroneous)) |- update -> pre_Out := erroneous;
  In1  = correct and In3 = correct
  and not (pre_Out = correct) |- update -> pre_Out := correct;
 assert
  Out = pre_Out;
edon
```

6.3.2 Modèle d'architecture matérielle

Nous ne détaillons pas le modèle d'architecture matérielle qui est réalisé selon les principes décrits dans le chapitre 3.

6.4 Exigences de sûreté de fonctionnement du SdT

Les différentes défaillances considérées sur les composants de ce système peuvent, lorsqu'elles sont combinées, entraîner la perte du contrôle de l'appareil. Dans cette situation, il est permis au pilote de reprendre le contrôle de son appareil afin d'éviter le « *crash* ». Pour que tout se déroule correctement, il est impératif que le pilote puisse connaître l'état de son système pour ainsi déceler d'éventuels dysfonctionnements.

Ainsi, partant de ces suppositions, différentes exigences sont dérivées pour que les missions puissent se dérouler sans problèmes. Les exigences considérées sont réparties suivant leurs degrés de criticité.

Exigences Catastrophiques :

- **« Pitch erroné non détecté »** .
  Nous nous intéresserons à l'absence d'alarme en cas de Pitch erroné. Cette situation est très critique car en mode de pilotage automatique, cette alarme est le seul moyen d'éviter un crash. Le pilote doit toujours être informé de l'état du SdT car en cas de défaillance dans le système, il doit pouvoir reprendre le pilotage pour sécuriser l'appareil. La formalisation de cette situation redoutée dans le modèle SdT est :
  `Pitch.Out = erroneous & Alarm.Out = false`
- **« Manoeuvre de Dégagement erronée »** .
  La manoeuvre de dégagement nécessite de disposer d'un angle de roulis correct. Si l'angle de roulis est erroné la manoeuvre peut conduire à la perte de l'appareil. La formalisation de cette situation redoutée dans le modèle SdT est :
  `Pitch.Out = erroneous & Alarm.Out = true & Roll.Out = erroneous`

Exigences Majeures :

– « **Dégagement intempestif** » . Cette situation apparaît lorsque l'alarme se déclenche alors que l'ordre de tangage est calculé correctement. Cette situation est redoutée puisque le dégagement peut gêner le pilote et l'empêcher de terminer sa mission correctement. La formalisation de cette situation redoutée dans le modèle SdT est :
`Pitch.Out = correct & Alarm.Out = true`

Exigences Mineures :

– « **Perte du SdT** » : La perte du système SdT n'est pas une situation à risque mais nous ne souhaitons pas que cette situation apparaisse trop souvent. Nous voulons donc identifier l'ensemble des défaillances qui causent l'arrêt de ce pilotage automatique. La formalisation de cette situation redoutée dans le modèle SdT est :
`Pitch.Out = lost`

Pour chacune de ces exigences, nous associons un noeud observateur en AltaRica et nous effectuons les vérifications correspondantes.

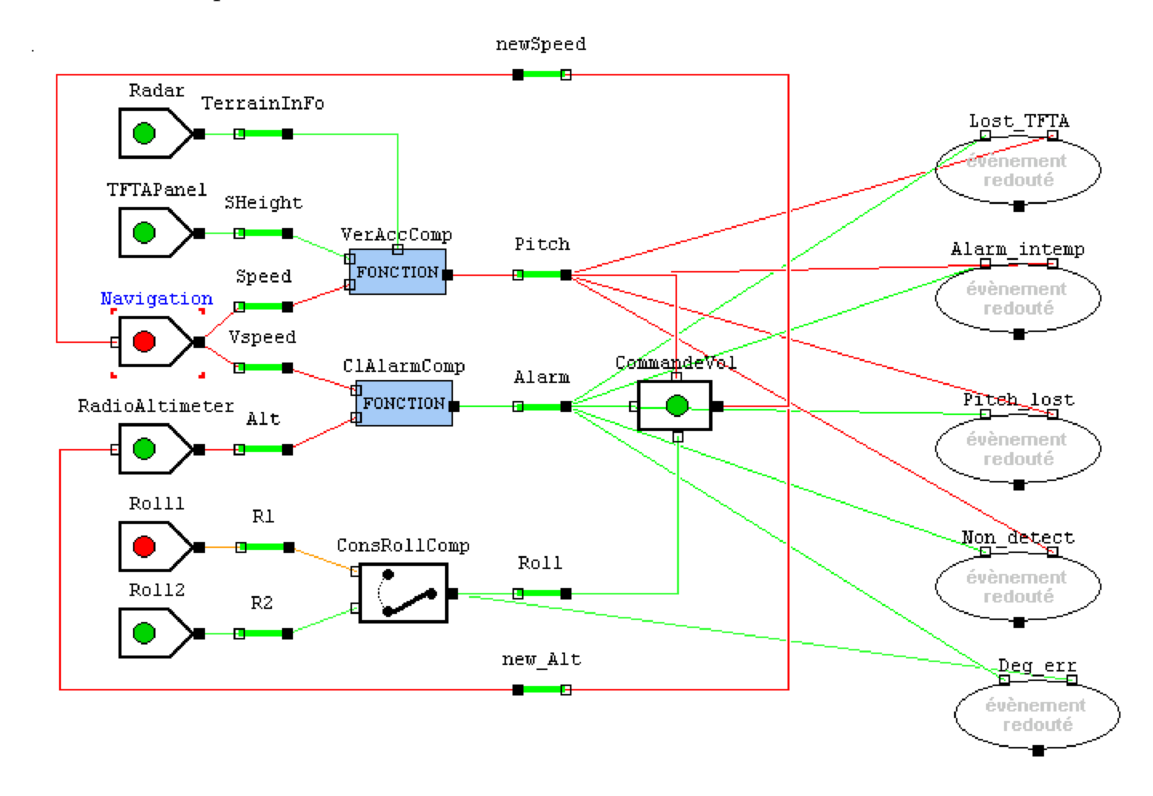


Fig. 6.7 – Observateurs associés aux situations redoutées

6.5 Identification des indépendances

L'outil Cecilia™ OCAS permet en plus de la modélisation, de générer les différents scénarios menant aux situations redoutées citées précédemment. Voyons plus en détail les différents scénarios. Pour la génération des scénarios menant aux situations redoutées, il ne faut ni considérer les composants matériels ni les composants fonctionnels représentant la communication (les *Bus*) car leur rôle dans l'architecture n'intervient que pour simuler des défaillances matérielles. Or, à ce stade de l'analyse, nous ne souhaitons pas considérer ce type de défaillances. Nous souhaitons juste nous assurer que l'architecture modélisée respecte bien les exigences qualifiées CAT et MAJ prédéfinies précédemment .

Il faut préciser que le système étudié est un système militaire qui n'est pas soumis à des exigences de sécurité aussi élevées qu'un avion civil. Aussi il est communément admis de « dégrader d'un cran » le niveau des exigences de sécurité. Ceci signifie que l'exigence militaire associée à une situation redoutée classée Catastrophique est « le taux de défaillance doit être inférieur à 10^{-7} par heures de vol et seules des combinaisons d'au moins deux défaillances peuvent conduire à cette situation ».

1. **« Pitch erroné non détecté »** : Pas d'alarme de dégagement lorsque l'angle de tangage est erroné.

   Dans l'état actuel de modélisation, les différents scénarios qui nous mènent à cette situation sont les suivants :

   ```
   /*Number of minimal cuts : 5*/

     'Non_detect.Out.true' :

   ('ClAlarmComp.fail_lost' & 'Navigation.fail_error');
   ('ClAlarmComp.fail_lost' & 'Radar.fail_error');
   ('ClAlarmComp.fail_lost' & 'TFTAPanel.fail_error');
   ('ClAlarmComp.fail_lost' & 'VerAccComp.fail_error');
   ('ClAlarmComp.fail_lost' & 'Roll1.fail_error' & 'Roll2.fail_error');
   ```

   Tous ces scénarios comportent au moins deux défaillances, dont la défaillance de la fonction de calcul de l'alarme de dégagement. Étant donné que cette situation est considérée comme Catastrophique, il ne faut pas que les scénarios impliquent moins de deux défaillances. Il faut donc que les composants qui sont impliqués dans un scénario de deux défaillances soient indépendants. Nous dérivons donc les hypothèses d'indépendance suivantes :

   ```
   idpt(ClAlarmComp, Navigation),
   idpt(ClAlarmComp, Radar),
   idpt(ClAlarmComp, TFTAPanel,
   idpt(ClAlarmComp, VerAccComp)
   ```

   Le dernier scénario est particulier puisqu'il est de taille 3, il suffit donc que deux parmi les trois composants impliqués dans ce scénario soient indépendant, nous obtenons :

   ```
   idpt(ClAlarmComp, Roll1) ou idpt(ClAlarmComp, Roll2) ou idpt(Roll1, Roll2)
   ```

2. **« Dégagement erroné »** : Cette situation redoutée correspond à la production d'un ordre erroné, d'une alarme et d'un roulis erroné. L'alarme de dégagement qui découle de cette situation peut conduire à la perte de l'appareil. Les différentes défaillances qui nous mènent à cette situation sont les suivantes :

   ```
   /*Number of minimal cuts : 1*/

     'Erroneous_deg.Out.true' :

   ('Navigation.fail_error' & 'Roll1.fail_error' & 'Roll2.fail_error');
   ```

   Le seul scénario qui mène à un dégagement dangereux est la combinaison de 3 défaillances. Afin d'éviter qu'une seule défaillance matérielle n'engendre cette situation, il faut rendre au moins deux défaillances indépendantes parmi les trois.

   De plus, il est possible d'assurer un niveau de sûreté supplémentaire, en considérant simplement que chaque défaillance du scénario doit être indépendante des autres. Ce nouveau choix nous garantit qu'au moins trois défaillances matérielles distinctes sont nécessaires pour rencontrer ce dégagement dangereux. Les nouvelles hypothèses indépendances dérivées sont donc les suivantes :

```
idpt(Navigation,Roll1)
idpt(Navigation,Roll2)
idpt(Roll1,Roll2)
```

3. « **Alarme intempestive** » .
   Les résultats obtenus sont :

```
/*Number of minimal cuts : 3  */

   'Alarm_intemp.Out.true' :

 ('ClAlarmComp.fail_error');
 ('RadioAltimeter.fail_lost');
 ('RadioAltimeter.fail_error');
```

   Ces trois scénarios n'apportent pas de nouveaux éléments à l'ensemble *idpt* puisqu'il sont tous constitués d'une seule défaillance.

Par l'analyse des différents scénarios menant à des situations que nous souhaitons éviter ou en s'assurant qu'il existe au moins deux défaillances matérielles y menant, nous avons construit l'ensemble *idpt* identifiant l'ensemble des fonctions à placer sur des ressources matérielles indépendantes. En effet, en plaçant chaque fonction de chaque couple de défaillances constituant l'ensemble *idpt* sur des ressources matérielles, nous garantissons que l'allocation des fonctions sur les ressources n'engendre pas de panne unique nous menant aux situations redoutées.

6.6 Génération des contraintes d'allocation

Les hypothèses d'indépendance à considérer pour notre *SdT* sont donc les suivantes :

```
idpt_ClAlarmComp_Navigation = 1;
idpt_ClAlarmComp_Radar = 1;
idpt_ClAlarmComp_TFTAPanel= 1;
idpt_ClAlarmComp_VerAccComp = 1;

idpt_Navigation_Roll1 = 1;
idpt_Navigation_Roll2 = 1;

idpt_Roll1_Roll2 = 1;
```

Le système basé sur les contraintes d'allocations est construit à partir des hypothèses d'indépendance et de l'ensemble des contraintes définies dans le chapitre 4.4. Chaque contrainte définie, doit être appliquée à notre système pour obtenir le système de contraintes à résoudre. Afin d'automatiser la création du système, un fichier de description de l'architecture fonctionnelle est nécessaire. Ce fichier ne comporte que la description des connexions entre les fonctions. En effet, grâce à la seule information des connexions, nous pouvons en déduire la liste des fonctions du système puisque chaque fonction du système utilise ou produit au moins une donnée et l'envoie à une autre fonction.

```
//Connexions fonctionnelles ..
f_cnx_Radar_VerAccComp = 1;
f_cnx_TFTAPanel_VerAccComp = 1;
f_cnx_Navigation_VerAccComp = 1;

f_cnx_Navigation_ClAlarmComp = 1;
```

```
f_cnx_RadioAltimeter_ClAlarmComp = 1;

f_cnx_Roll1_ConsRollComp = 1;
f_cnx_Roll2_ConsRollComp = 1;

f_cnx_VerAccComp_CommandeVol = 1;
f_cnx_ClAlarmComp_CommandeVol = 1;
f_cnx_ConsRollComp_CommandeVol = 1;

f_cnx_CommandeVol_Navigation = 1;
f_cnx_CommandeVol_RadioAltimeter = 1;
```

Le fichier de description est ensuite complété par les hypothèses d'indépendance précédemment identifiées et par les directives d'allocations propres au système. Pour notre système, nous souhaitons allouer la fonction de commande de vol à une ressource particulière, car nous supposons que cette fonction nécessite une ressource ayant certaines caractéristiques. Nous supposons ainsi que la ressource `R3` possède des caractéristiques qui lui sont propres et répond aux besoins de la fonction Commande de vol.

Une dernière information importante pour construire le système est le type des ressources mises à disposition pour l'allocation. Le nombre de ressources n'a pas vraiment d'importance, car comme nous le verrons plus tard, un critère d'optimisation du système de contrainte, consistant à choisir la solution proposant le minimum de ressources ou bien proposant le minimum de connexion à utiliser, est mis en oeuvre.

Pour notre système, sachant que l'exigence permettant d'éviter la situation d'un *Dégagement erroné* impose la ségrégation (indépendance) de trois fonctions, il est donc nécessaire d'allouer le *SdT* sur au moins trois ressources matérielles (que nous nommerons `R1`, `R2`, `R3`).

Ainsi, le fichier de description permettant de construire le système de contrainte est enrichi par les informations suivantes :

```
// Hypothese independance
idpt_ClAlarmComp_Navigation = 1;
idpt_ClAlarmComp_Radar = 1;
idpt_ClAlarmComp_TFTAPanel= 1;
idpt_ClAlarmComp_VerAccComp = 1;

idpt_Navigation_Roll1 = 1;
idpt_Navigation_Roll2 = 1;

idpt_Roll1_Roll2 = 1;

// Directives d'allocation
alloc_CommandeVol_R3 = 1;

#Ress: R1 R2 R3
```

6.7 Recherche et visualisation d'allocation

6.7.1 Recherche d'une allocation

La recherche d'une allocation est rendue automatique par la création d'une application. En effet, la recherche d'une allocation correspond à une solution du système de contraintes. Cette recherche consiste donc à résoudre le système et à extraire les résultats pour visualiser l'allocation trouvée. Cette méthode est constituée de plusieurs étapes, une première consiste à transformer le fichier de description de notre système (créé dans la section précédente) en un fichier décrivant le système de contraintes utilisable par le solveur SatZoo.

Le formalisme utilisé par SatZoo est le formalisme ILP[2]. Pour une résolution du système par ce solveur, l'ensemble des contraintes créé doit être en inéquations linéaires acceptées par le solveur.

La partie description du système ne correspond pas à un ensemble de contraintes, mais elle permet de définir l'ensemble des variables qui vont être manipulées par le solveur. Étant donné que SatZoo n'accepte que des inéquations, chaque équation utilisée pour la description du système doit être adapté en un ensemble d'inéquations pour ainsi être utilisé par SatZoo. Par exemple, pour décrire une connexion entre la fonction de calcul d'alarme de dégagement et les commandes de vol, il faut définir la variable :

```
f_cnx_ClAlarmComp_CommandeVol  >=  1
```

L'application de cette règle sur l'ensemble des variables utilisé pour la description de notre système permet ainsi de définir un premier ensemble d'inéquations. Les inéquations à ajouter pour obtenir un premier système à résoudre sont celles identifiées pour respecter les indépendances. En effet, les analyses précédentes ont permis de mettre en évidences plusieurs indépendances entre fonctions à respecter pour tenir certaines de nos exigences. La description du système identifie les indépendances à respecter par un ensemble de variables définies comme suit :

```
idp_Fonction1_Fonction2  >=  1
```

Pour chaque couple de fonctions présentes dans une contrainte d'indépendance, il faut s'assurer que les deux fonctions du couple ne sont pas allouées sur une même ressource. Par exemple dans le cas de l'indépendance entre la fonction de calcul de l'alarme de dégagement et la fonction de Navigation, la variable permettant la description est la suivante :

```
idp_ClAlarmComp_Navigation >= 1;
```

Les inéquations qui en découlent pour s'assurer que la fonction de calcul de l'alarme et la fonction Navigation ne sont pas allouées sur une même ressource sont les suivantes (en considérant qu'il existe trois ressources R1, R2, R3) :

```
alloc_ClAlarmComp_R1 + alloc_Navigation_R1 + idp_ClAlarmComp_Navigation <=2;
alloc_ClAlarmComp_R2 + alloc_Navigation_R2 + idp_ClAlarmComp_Navigation <=2;
alloc_ClAlarmComp_R3 + alloc_Navigation_R3 + idp_ClAlarmComp_Navigation <=2;
```

Le système de contraintes est ensuite enrichi par l'ensemble des contraintes définies dans 4.4 et appliqué aux différentes variables du système pour construire le système de contraintes globales s'appliquant à notre système. Une fois le système de contraintes complet, nous utilisons l'outil SatZoo pour traiter sa résolution. Lorsqu'une solution existe, l'outil SatZoo renvoie cette dernière sous forme d'une valuation de l'ensemble des variables du système correspondant à l'allocation trouvée.

6.7.2 Visualisation de l'allocation

Pour simplifier la représentation du système, une application dédiée à la visualisation des allocations proposées par SatZoo a été implémentée.

[2]Integer Linear Programming

Cette application permet de visualiser rapidement une solution d'allocation des fonctions sur les ressources matérielles. Lorsque SatZoo propose plusieurs solutions pour un même système, l'application permet de faire défiler ces différentes allocations possibles afin de choisir celle présentant le plus d'intérêt. En effet, suivant le critère d'optimisation choisi pour orienter le choix de la meilleure solution, il se peut que plusieurs solutions aient la même optimisation, mais possèdent des allocations différentes. Il est important que la sélection définitive d'une allocation soit réservée aux personnes en charge de la réalisation du système. Par conséquent lorsque plusieurs solutions sont disponibles, il faut les préserver pour de nouvelles analyses manuelles de sélection.

Dans l'état actuel de la résolution de notre système, SatZoo nous propose une solution et donc une allocation possible pour notre système. La représentation faite par SatZoo de la solution est la suivante :

```
MODEL:
alloc_Navigation_R1
alloc_Roll1_R2
alloc_CommandeVol_R3
alloc_Roll2_R3
alloc_ClAlarmComp_R3
alloc_ConsRollComp_R3
alloc_Radar_R3
alloc_RadioAltimeter_R3
alloc_TFTAPanel_R3
alloc_VerAccComp_R3
-u_cnx_R1_R2 u_cnx_R1_R3 u_cnx_R2_R3
used_R1 used_R2 used_R3
```

Le résultat présenté ci-dessus peut être interprété comme suit :

- les variables de la forme `alloc_Fonction_Ressource` dont la valeur de vérité est vraie indiquent que `Fonction` est allouée à la `Ressource`. Par exemple, les fonctions `CommandeVol` et `Roll2` sont allouées à la ressource `R3`.
- les variables `used_Ressource` dont la valeur de vérité est vraie indiquent que la ressource est utilisée. Dans le cas précédent,les trois ressources sont utilisées.
- les variables `u_cnx_Ressource1_Ressource2` dont la valeur de vérité est fausse (elle sont précédées du signe « - ») indiquent que la connexion entre deux ressources n'est pas utilisée. Par exemple, la variable `-u_cnx_R1_R2` signifie que la connexion entre les ressources `R1` et `R2` n'est pas utilisée et par conséquent nous pouvons en déduire qu'il n'est pas nécessaire d'avoir une liaison entre ces deux ressources.
- les variables `u_cnx_Ressource1_Ressource2` dont la valeur de vérité est vraie décrivent les orientations à respecter pour construire l'architecture matérielle.

Afin de mieux représenter le résultat obtenu, il est possible de visualiser cette allocation à l'aide d'un outil développé dans ce travail. La représentation de l'allocation par l'outil est donnée par la figure suivante :

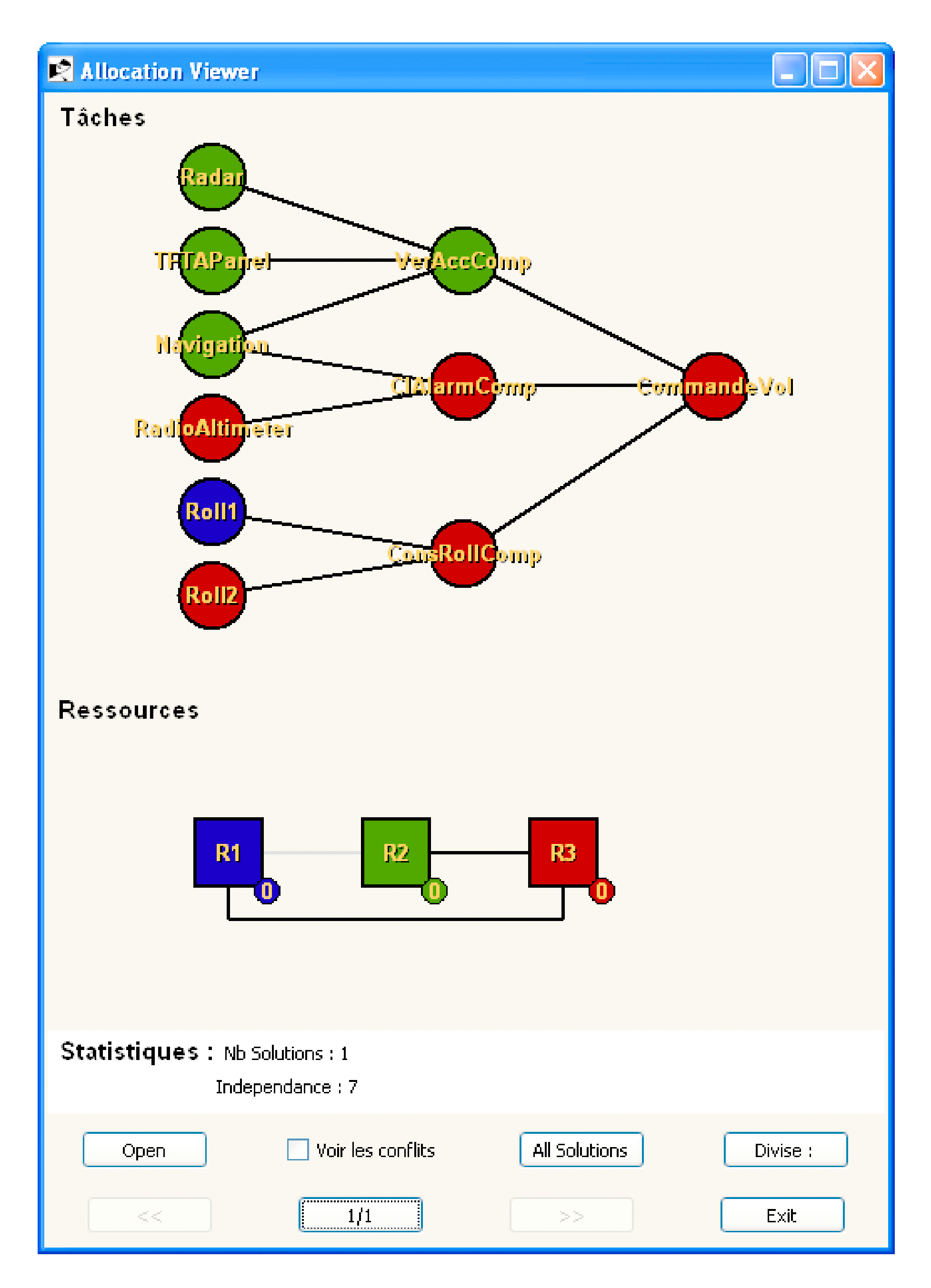
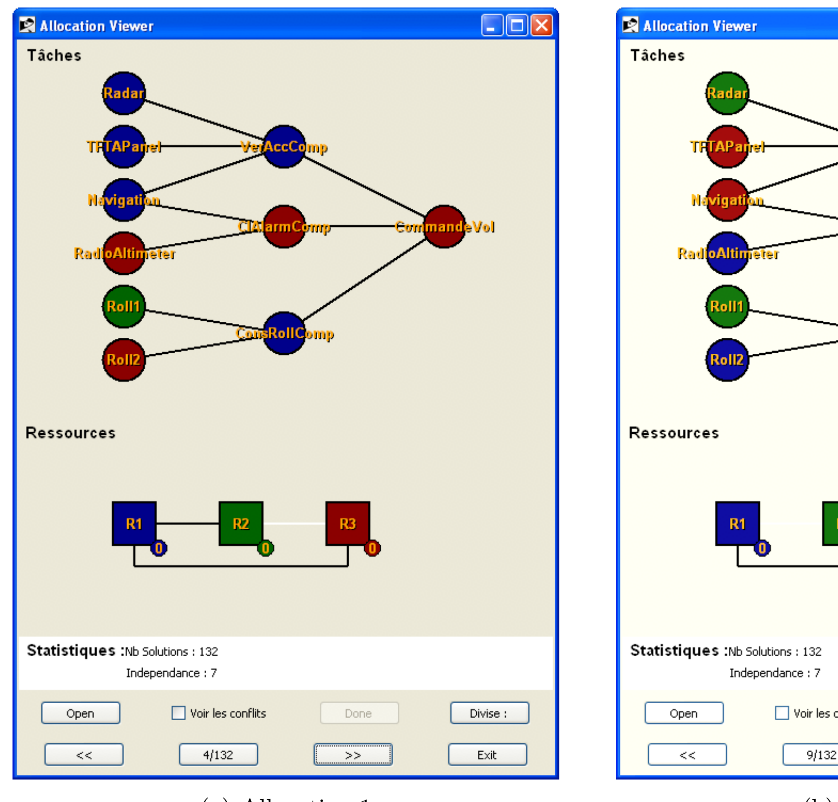


Fig. 6.8 – AllocViewer pour la visualisation des résultats

Grâce à cet outil, il est possible de visualiser les allocations possibles une à une afin de sélectionner celle qui possède le plus d'intérêt pour le concepteur de l'architecture. En effet, par l'appui sur le bouton « All Solution », l'outil retourne l'ensemble des solutions ayant le même critère optimisé. Ensuite, par des appuis successifs sur les « flèches », l'utilisateur peut parcourir les solutions (cf. figure 6.9).



(a) Allocation 1 (b) Allocation n

Fig. 6.9 – Autres solutions possibles du système

Suivant les choix établis à partir de nouveaux critères d'optimisation, le concepteur peut préférer une allocation plutôt qu'une autre. Par exemple comme visible dans la figure précédente, l'allocation n permet une répartition plus équitable des fonctions sur les ressources.

Une fois obtenue une allocation, il est possible d'étendre le modèle de l'architecture fonctionnelle avec des synchronisations qui représentent l'effet des défaillances matérielles sur les composants fonctionnels. Le modèle obtenu peut être utilisé pour affiner l'analyse. Par exemple, en cherchant à évaluer l'impact quantitatif des défaillances matérielles sur la probabilité d'occurrence d'une situation redoutée. Ceci a été réalisé pour le système SdT et décrit dans l'article [SB07].

6.8 Bilan

Nous avons développé un outil permettant de générer automatiquement et de visualiser un ensemble d'allocations de fonctions sur des ressources satisfaisant un ensemble de contraintes. Cet outil utilise des contraintes qui sont dérivées à partir de la modélisation AltaRica de l'architecture fonctionnelle d'un système.

Cette approche a été étendue pour traiter avec plus de détail les architectures de réseaux distribués au sein des avions civils. L'approche a été appliquée par l'ONÉRA et l'IRIT à l'étude du placement des fonctions d'un système de détection d'incendie sur une architecture d'un avion de type A380 [BBC+08].

CHAPITRE 7

SYSTÈME HYDRAULIQUE D'UN AVION DE TYPE A320

Nous appliquons au système hydraulique d'un avion civil la technique de modélisation et validation de l'allocation des équipements dans les zones d'un avion qui a été présentée dans le chapitre 3. Cette technique de modélisation se repose sur le couplage de modèles décrivant le même système mais construits à l'aide de langages différents : AltaRica pour la sûreté de fonctionnement et IRIS[Air00] pour l'installation des équipements au sein de l'avion.

SOMMAIRE

7.1 Démarche utilisée

La démarche que nous avons définie dans le projet ISAAC[1] consiste à injecter les résultats des analyses d'un risque particulier dans le modèle AltaRica d'architecture fonctionnelle. Actuellement, l'analyse d'un modèle pour un risque particulier (par exemple pour un risque d'éclatement moteur) consiste à trouver l'ensemble des éléments impactés pour chaque trajectoire de débris ejecté lors de l'éclatement. Une fois ces résultats obtenus, l'analyste doit évaluer si ces trajectoires conduisent à des situations invalidant les exigences de sûreté de fonctionnement.

Cette méthodologie est décomposée en plusieurs étapes :

1. Construction du modèle d'architecture fonctionnelle avec AltaRica et du modèle d'installation avec IRIS pour le système étudié
2. Allocation des fonctions sur les composants du modèle d'installation
3. Calcul des ensembles de composants impactés par un éclatement pneu ou moteur.
4. Extension du modèle d'architecture fonctionnelle avec des événements représentant la défaillance simultanée de toutes les fonctions allouées sur un composant impacté par un éclatement pneu ou moteur.
5. Vérification de la tenue des exigences de sûreté de fonctionnement à l'aide du modèle étendu.

Dans la suite de ce chapitre, nous verrons comment appliquer cette méthodologie sur le cas d'étude utilisé durant le projet ISAAC.

7.2 Présentation du système

Le système étudié correspond au système hydraulique de l'A320.

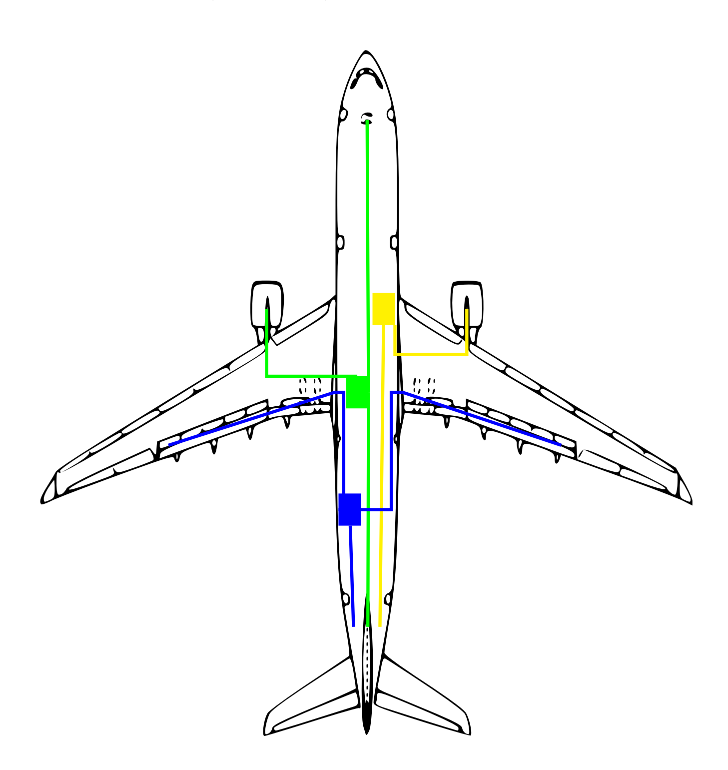


Fig. 7.1 – Système Hydraulique de l'A320

[1]Improvement of Safety Activities on Aeronautical Complex systems

But : Le rôle du système hydraulique est de fournir la puissance hydraulique avec le niveau de sûreté adéquat à la fois aux éléments permettant le contrôle de l'avion en vol (servo-commandes, becs et volets, gouvernes, etc.) et aux éléments utilisés au sol (train d'atterrissage, système de freinage, etc.).

7.2.1 Description :

Le système hydraulique de l'A320 est composé des éléments suivants :

- Des pompes qui génèrent la puissance hydraulique nécessaire aux différents consommateurs. Il y a trois types de pompes :
  - celles alimentées par un courant électrique appelées EMP (Electric Motor Pump),
  - celles alimentées par les réacteurs appelées EDP (Engine Driven Pump) et
  - celle alimentée par le dispositif de secours appelée RAT (Ram Air Turbine).
- De trois lignes de distribution (Green, Blue et Yellow) qui transmettent la puissance hydraulique des réservoirs vers les consommateurs.
- De vannes de priorité qui permettent d'isoler les consommateurs non prioritaires et d'alimenter uniquement les éléments critiques.
- D'un PTU (Power Transfert Unit) qui, lorsqu'il y a une différence de pression entre les systèmes Jaune et Vert, permet d'alimenter le système ne fournissant pas de puissance en utilisant la puissance hydraulique du système qui en délivre le plus.

Le système bleu possède deux pompes : une pompe électrique (que l'on notera EMPb) et la RAT. En cas de perte des deux réacteurs, ou bien en cas de perte totale de la génération électrique un système de secours appelé RAT, situé sous l'avion, se déploie automatiquement et alimente en puissance hydraulique le circuit bleu.

Le système vert possède une pompe mécanique (notée EDPg) liée au réacteur n°1. En cas de défaillance de cette pompe et uniquement si la vanne de priorité du système jaune est ouverte, le PTU alimente automatiquement le circuit vert en fournissant une partie de la puissance du circuit jaune.

Le circuit jaune possède deux pompes : une pompe électrique (notée EMPy) ainsi qu'une pompe mécanique alimentée par le réacteur n°2 (notée EDPy). De même que pour le circuit vert, en cas de panne de ces deux pompes et si la vanne de priorité du circuit vert est ouverte, le PTU alimente le circuit jaune en fournissant une partie de la puissance du circuit vert.

En cas de panne de réacteur, la pompe mécanique associée ne fonctionne plus. En cas de défaut d'alimentation électrique des pompes électriques, l'APU[2] peut fournir de l'énergie électrique à ces pompes.

[2] Auxiliary Power Unit

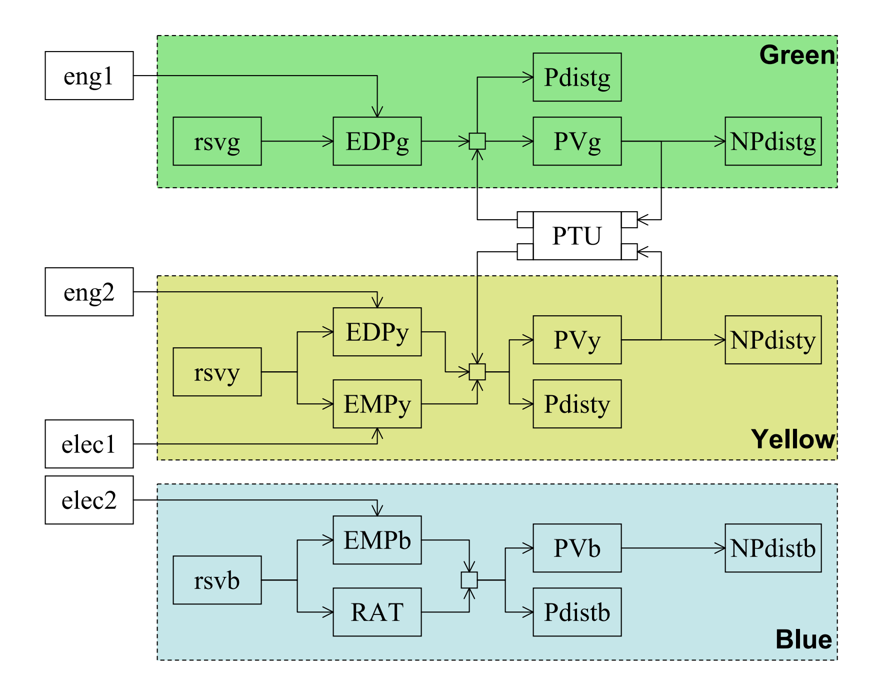


FIG. 7.2 – Architecture du système Hydraulique

7.2.2 Les exigences à vérifier

Les différentes exigences que nous souhaitons valider sur ce système concernent les différentes combinaisons possibles de perte des lignes hydrauliques. Nous considérons plusieurs situations à risques :

- « La perte totale de la puissance hydraulique » est une situation considérée du niveau le plus critique, car elle concerne l'ensemble des lignes hydrauliques. Étant donné que de nombreux systèmes sont en relation directe avec cette source de puissance, la perte de l'ensemble des lignes hydrauliques pourrait engendrer une situation hautement dangereuse. Ainsi pour qualifier cette situation, nous choisissons la catégorie la plus critique : CAT. Comme vu dans le chapitre 2.1.2, l'exigence qualitative qui en découle est qu'il ne faut pas qu'une panne simple ou double nous mène à cette situation. Et l'exigence quantitative fixe un taux d'occurrence inférieur à 10^{-9} par heure de vol pour cette situation.

- « La perte partielle de la puissance hydraulique » correspond à la perte d'au moins deux lignes parmi les trois possibles. En effet, en cas de perte d'au moins deux lignes, les différentes fonctions importantes doivent être alimentées par la dernière ligne disponible. Nous devons donc nous assurer que les différentes logiques de reconfiguration permettent en cas de perte de plusieurs lignes de garantir l'alimentation en puissance des éléments critiques de l'avion. Cette exigence est qualifiée MAJ car elle n'entraîne pas la perte de l'avion, mais assure le fonctionnement minimal nécessaire à la protection de l'appareil et donc de ses passagers. L'exigence qualitative que nous associons à cette exigence est qu'il ne faut pas qu'une panne simple nous mène à cette situation. Et l'exigence quantitative fixe un taux d'occurrence inférieur à 10^{-5} par heure de vol pour cette situation.

- « La perte d'une simple ligne hydraulique » n'est pas une situation critique, mais étant donné qu'elle implique une partie de la source en puissance de l'avion, nous souhaitons maîtriser les différents scénarios de pannes (indépendamment du nombre de défaillances) menant à la perte

d'une ligne hydraulique. Cette situation est donc considérée comme MIN. Nous n'associons pas d'exigence qualitative à cette situation. En revanche nous fixons un taux d'occurrence inférieur à 10^{-3} par heure de vol pour cette situation.

7.3 Modélisation de l'architecture fonctionnelle du système hydraulique

Les choix de modélisation du système sont influencés par les types d'analyses que nous souhaitons faire sur ce dernier. En effet, l'idée première était de combiner deux outils de modélisation (IRIS et OCAS) afin de proposer une analyse complémentaire. La modélisation du système dédiée aux analyses d'impact étant déjà effectuée, il est important d'extraire certaines informations utiles pour cette nouvelle modélisation (modélisation fonctionnelle). Il peut s'agir par exemple des éléments modélisés, de leur nombre, de leurs connexions, etc. Le choix des éléments considérés du modèle déterminent le niveau de détail obtenu pour le modèle fonctionnel.

Une fois tous les composants importants identifiés, il faut leur associer un comportement suffisamment abstrait afin d'obtenir un modèle facilement analysable (pour éviter l'explosion combinatoire) tout en conservant un comportement proche du système réel.

7.3.1 Modélisation des éléments du système Hydraulique

La modélisation de ce système doit permettre de visualiser l'influence d'une panne sur le système global pour ainsi observer la réaction du système face à un composant défaillant. Pour cela, nous associons aux différents composants un comportement propre face à une ou plusieurs défaillances. Ensuite, nous devons identifier les informations qui vont circuler dans notre système. En effet pour pouvoir étudier la propagation des défaillances dans le système, il faut identifier les informations importantes qui vont transiter entre les composants et ainsi influencer leurs comportements.

Dans le système hydraulique étudié, deux caractéristiques importantes des connexions doivent être prises en compte :

- Tout d'abord, la première caractéristique est la présence ou l'absence de puissance dans une connexion. Cette caractéristique est importante, car nous étudions un système dont l'objectif est de fournir une puissance à un ensemble de composants utilisateurs. Par exemple, lorsqu'une source de puissance subit une défaillance, alors elle ne peut plus fournir cette puissance et par conséquent tous les composants connectés à cette source doivent être informé de l'absence de puissance par leur connexion respective.

  Cette caractéristique est représentée par un booléen associé à chaque connexion des composants. L'absence de puissance est modélisée par la valeur fausse et inversement la présence de puissance correspond à la valeur vraie.
- La seconde caractéristique importante des connexions est la présence ou l'absence de fuite. En effet, le système est sujet à des défaillances représentant différentes fuites qui vont influencer la puissance générale fournie par le système. Comme la puissance transitant dans les connexions varie suivant la pression du fluide dans ces connexions, il faut informer les composants connectés de la présence d'une fuite. Par exemple, en cas de présence de fuite, l'information doit parvenir à la source du fluide pour modéliser la baisse de niveau de fluide.

  De même que précédemment, cette caractéristique est représentée par un booléen : vrai en cas de présence d'une fuite et faux dans le cas inverse.

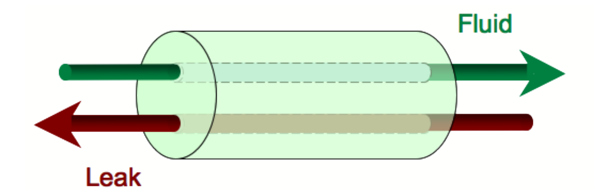


Fig. 7.3 – Modélisation des informations circulant dans le système

Pour la suite de la modélisation, nous considérons que le sens dans lequel le fluide est amené à se déplacer représente le sens normal (`nom`) et inversement, le sens permettant de caractériser une fuite (`leak` sur la figure 7.3) représente le sens inverse (`inv`). La modélisation d'une telle information en AltaRica passe par la création d'un nouveau type. Ce type permet, lorsqu'il est associé à un port de connexion d'un composant, de manipuler les deux sens tout en utilisant qu'une seule connexion. En effet, si nous considérons par exemple un composant n'ayant qu'une seule connexion (`O`) et que cette dernière représente une production de fluide, alors le sens normal de la connexion représente l'information de présence de fluide et le sens inverse représente l'information de présence d'une fuite.

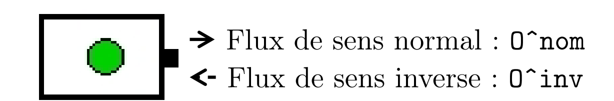


Fig. 7.4 – Modélisation des informations circulant dans le système en AltaRica

Le code AltaRica permettant de modéliser ce type d'information circulant entre les composants est le suivant :

```
node compo
 flow
  O^nom : bool : out ;
  O^rev : bool : in ;
 ...
edon
```

Lorsque les informations caractérisant les connexions entre les composants sont identifiées et modélisées, il reste à modéliser les différents composants eux-mêmes :

1. **Les sources d'énergie** : les moteurs (Eng1, Eng2), les alimentations électriques (Elec1, Elec2). Ces sources permettent d'alimenter en électricité ou en puissance mécanique les différents pompes du système hydraulique. Leur rôle consiste donc à produire de l'énergie (dans ce cas la sortie `O` est vraie) lorsque qu'ils ne sont pas en présence d'une défaillance, et de ne rien produire dans le cas contraire (la sortie `O` est fausse dans ce cas). Le comportement souhaité en présence de la défaillance `fail_Loss`, est que l'état interne `S` du composant change, empêchant ainsi la production d'énergie. Ce comportement peut être représenté par l'automate suivant :

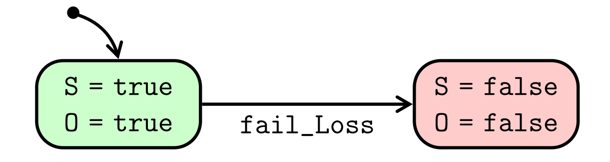
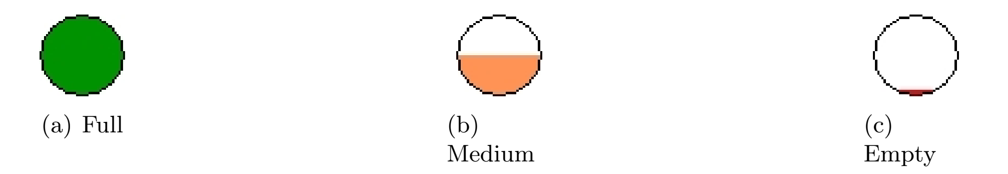


Fig. 7.5 – Automate du comportement d'un réservoir

La transcription en AltaRica d'un tel comportement peut s'exprimer de la façon suivante :

```
node Source
 flow
   O:bool:out;
 state
   S:bool;
 event
   fail_Loss;
 trans
   S |- fail_Loss -> S := false;
 assert
   O = S;
 init
   S := true;
edon
```

2. **Les réservoirs** : Un réservoir est affecté à chaque ligne de distribution. Le comportement souhaité pour ce composant est qu'en présence d'une fuite sur la ligne hydraulique qu'il alimente, le réservoir doit se vider progressivement. Ces différentes étapes considérées sont représentées par la figure 7.6(a).



(a) Full (b) Medium (c) Empty

Fig. 7.6 – Les différents états d'un réservoir

Afin de modéliser correctement le « vidage progressif » du réservoir, l'événement `update` est ajouté. Il permet, en présence d'une fuite dans le système (qui est signalée par le fait que l'entrée `O^rev` est fausse), de modifier en deux étapes la variable d'état du composant (`S` passe de la valeur `full` à `medium` puis `empty`) pour visualiser l'évolution du contenu du réservoir. Nous considérons que le réservoir fournit du fluide (la sortie `O^nom` est vraie) tant qu'il n'est pas vide (la variable d'état `S` est différentede la valeur `empty`) .

De plus, à tout moment une défaillance `fail_Loss` peut provoquer le vidage instantanné du réservoir. Le graphe représentant un tel comportement peut être le suivant :

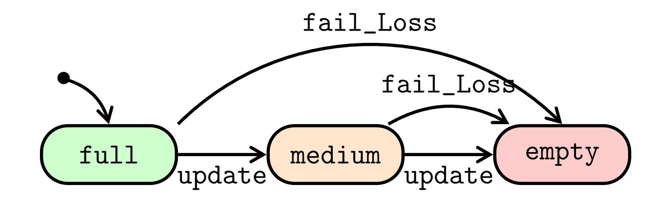


Fig. 7.7 – Automate du comportement d'un réservoir

Le code AltaRica correspondant au comportement du réservoir peut s'écrire de la façon suivante :

```
node rsv
  flow
    icone:[1,3]:private;
    O^nom:bool:out;
    O^rev:bool:in;
  state
    S:{full, medium, empty};
```

```
  event
    update,
    fail_Loss;
  trans
    (not (S = empty)) |- fail_Loss -> S := empty;
    ((O^rev = false) and (S = full)) |- update -> S := medium;
    ((O^rev = false) and (S = medium)) |- update -> S := empty;
  assert
    O^nom = not (S = empty);
  init
    S := full;
edon
```

3. **Les pompes** : elles sont électriques (EMPy et EMPb) et mécaniques (EDPg et EDPy) et ont le même comportement pour nos analyses : si une pompe est activée, qu'elle est alimentée en énergie (l'entrée `A` est vraie), qu'elle n'est pas en panne (la variable d'état `S` est vraie) et qu'il y a du fluide hydraulique (l'entrée `I^nom` est vraie) alors la pompe produit de la puissance hydraulique (la sortie `O^nom` est vraie) et sinon elle n'en produit pas. Une pompe peut subir le mode de défaillance perte (événement `fail_loss`), dans ce cas la pompe ne produit plus de puissance hydraulique.

   De plus, la pompe propage, dans tous les cas, la présence d'une fuite d'un composant en aval (entrée `O^rev` est vraie) aux composants connectés en amont (sortie `I^rev` est vraie).

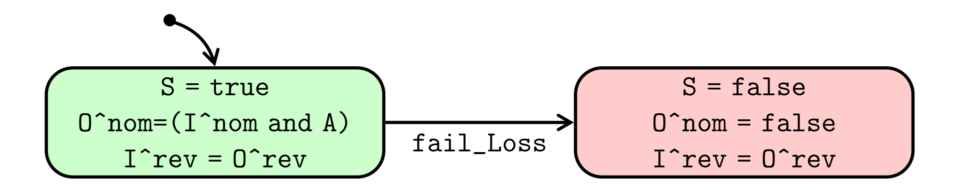


Fig. 7.8 – Automate du comportement d'une pompe

Le code AltaRica correspondant au comportement d'une pompe peut s'écrire de la façon suivante :

```
node pump
 flow
  A : bool : in ;
  O^nom : bool : out ;
  O^rev : bool : in ;
  I^nom : bool : in ;
  I^rev : bool : out ;
 state
  S : bool ;
 event
  fail_Loss ;
 init
   S := true ;
 trans
  S=true |- fail_Loss -> S:=false;
 assert
  O^nom = (I^nom and S and A);
  I^rev = O^rev;
edon
```

4. **Les vannes** de priorité ont pour objectif de couper la puissance hydraulique délivrée aux lignes de distribution non-prioritaires. Nous utilisons le même noeud AltaRica que lors de la modélisation d'une pompe en considérant que la vanne est ouverte lorsque le signal d'activation `A` est vrai et qu'elle est fermée sinon.

5. **Les consommateurs** sont de deux types : prioritaires et non prioritaires. Les consommateurs prioritaires ont comme leurs noms l'indiquent une priorité sur l'alimentation par rapport aux non prioritaires. Ils ont tous le même comportement à savoir : consommer de la puissance hydraulique, subir une défaillance de type fuite (événement `fail_leakage`) et, dans ce cas, envoyer une information de fuite (sortie `I^rev` est vraie).

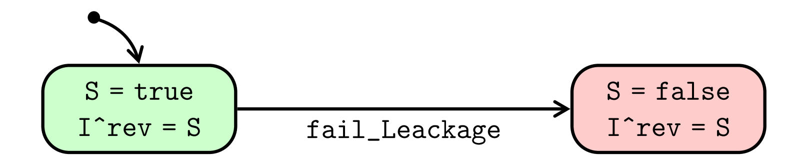


Fig. 7.9 – Automate du comportement d'un consommateur

La transcription en AltaRica de ce comportement peut s'exprimer de la façon suivante :

```
node dist
 flow
  I^nom : bool : in ;
  I^rev : bool : out ;
 state
  S : bool ;
 event
  fail_Leackage ;
 init
  S := true ;
 trans
  S = true |- fail_leackage -> S := false;
 assert
  I^rev = S;
edon
```

6. **Le PTU** est un des composants les plus complexes, car il doit observer l'état de deux lignes hydrauliques afin de pouvoir faire basculer de la puissance d'une première ligne vers une seconde s'il vient à détecter la perte de puissance dans la seconde ligne. Ce composant possède donc des entrées afin de récupérer de la puissance hydraulique (`I1^nom` représente la puissance hydraulique de la ligne 1, `I2^nom` représente la puissance hydraulique de la ligne 2) et des sorties pour la transmettre sur l'autre ligne (`O1^nom` représente la puissance hydraulique fournie à la ligne 1 par la ligne 2, `O2^nom` représente la puissance hydraulique fournie à la ligne 2 par la ligne 1). Il possède également des entrées (`A1` est vraie lorsqu'il y a de la puissance hydraulique sur la ligne 1 et `A2` est vraie lorsqu'il y a de la puissance hydraulique sur la ligne 2) permettant d'observer l'état des lignes hydrauliques et donc d'activer le mécanisme de transfert de puissance hydraulique.

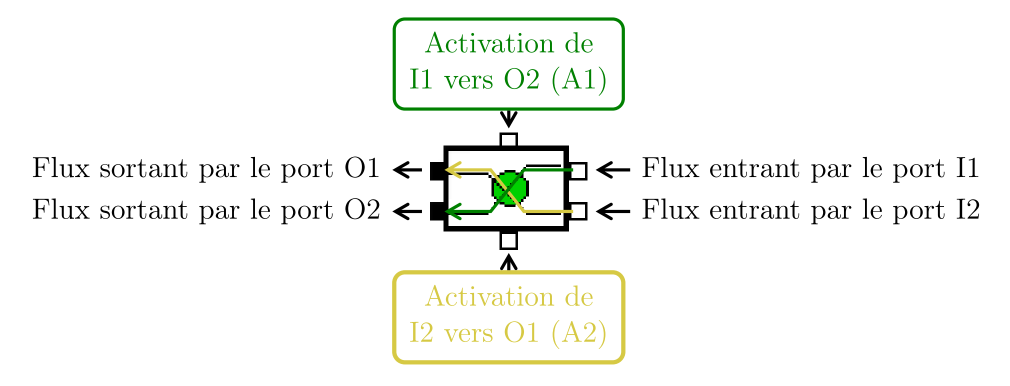


Fig. 7.10 – Rôle du PTU

Le comportement souhaité pour ce composant est décrit comme suit :

- lorsque les deux lignes délivrent de la puissance hydraulique ou lorsqu'aucune des deux lignes ne délivre de la puissance hydraulique alors le PTU ne doit pas être activé. Dans ce cas une fuite d'un consommateur d'une ligne n'a pas d'influence sur le niveau de fluide de l'autre ligne.
- Lorsque la ligne 2 ne délivre pas de puissance hydraulique et que la ligne 1 en délivre alors le PTU doit être activé. Dans ce cas, la puissance hydraulique de la ligne 1 alimente les consommateurs de la ligne 2. Dans ce cas, une fuite sur un consommateur de la ligne 2 influe sur le niveau de fluide de la ligne 1.
- Le comportement doit être similaire lorsque la ligne 1 ne délivre pas de puissance hydraulique et que la ligne 2 en délivre.
- Tant qu'il n'y a pas d'activation, le PTU ne doit transférer aucune puissance hydraulique.
- En présence d'un dysfonctionnement, le PTU ne fournit plus de puissance hydraulique d'une ligne à l'autre mais la propagation d'une ligne à l'autre de la baisse de niveau de fluide liée à la fuite d'un des consommateurs n'est pas arrêtée.

Pour modéliser ce comportement en AltaRica, nous utilisons de nouvelles variables d'états `P12_nom`, `P12_rev`, `P21_nom` et `P21_rev` qui déterminent l'activation du PTU. `P12_nom` est la variable d'état correspondant à l'activation du transfert de puissance hydraulique de la ligne 1 vers la ligne 2. En cas d'activation (`A1 = true and A2= false`), cette variable est mise à jour pour permettre le changement de mode (`P12_nom :=true` si `I1^nom :=true`). Cette mise à jour est effectuée par l'événement `update`. Le code AltaRica de la transition de ce comportement est donc :

```
trans
 S = true |- update -> P12_nom := (((not A2) and A1) and I1^nom)
```

De même la variable `P12_rev` est la variable d'état correspondant à l'activation de la propagation des fuites de la ligne 2 vers la ligne 1. En cas d'activation (`A1 = true and A2= false`), cette variable est mise à jour pour permettre le changement de mode (`P12_rev :=true` si `O2^rev :=true`).

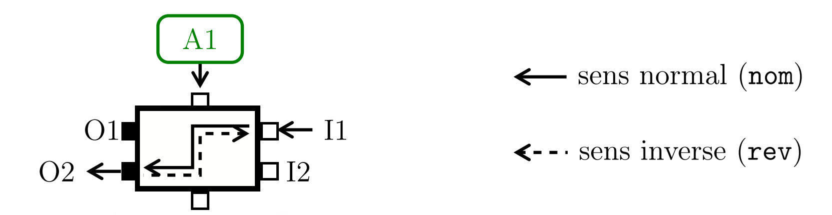


Fig. 7.11 – Comportement du PTU en cas d'activation A1

Comme présenté dans la figure 7.10, les valeurs des flux sortants du PTU sont fonctions de l'activation effectuée et par conséquent ils sont fonctions des variables d'état de ce composant.

En appliquant ces choix pour chaque variable d'état et pour les deux sens de propagation, le code AltaRica suivant formalise le comportement du PTU :

```
node PTU
 flow
  icone:[1,2]:private;
  I1^nom:bool:in;
  I1^rev:bool:out;
  O1^nom:bool:out;
  O1^rev:bool:in;
  A1:bool:in;
  A2:bool:in;
  O2^nom:bool:out;
  O2^rev:bool:in;
```

```
  I2^nom:bool:in;
  I2^rev:bool:out;
 state
  S:bool;
  P12_rev:bool;
  P12_nom:bool;
  P21_rev:bool;
  P21_nom:bool;
 event
  fail_Loss,
  update;
 trans
  S = true |- fail_Loss -> S := false;
  S = true |- update -> P21_nom := (((not A1) and A2) and I2^nom),
                        P12_nom := (((not A2) and A1) and I1^nom),
                        P21_rev := (((not A1) and A2) and O1^rev),
                        P12_rev := (((not A2) and A1) and O2^rev);
 assert
  O1^nom = (S and P21_nom),
  O2^nom = (S and P12_nom),
  I1^rev = P12_rev,
  I2^rev = P21_rev;
 init
  S := true,
  P12_rev := false,
  P12_nom := false,
  P21_rev := false,
  P21_nom := false;
edon
```

La composition de l'ensemble de ces composants permet de construire le modèle AltaRica du système hydraulique. Ce modèle est construit à l'aide de l'outil *Cécilia OCAS* et est représenté par la figure suivante :

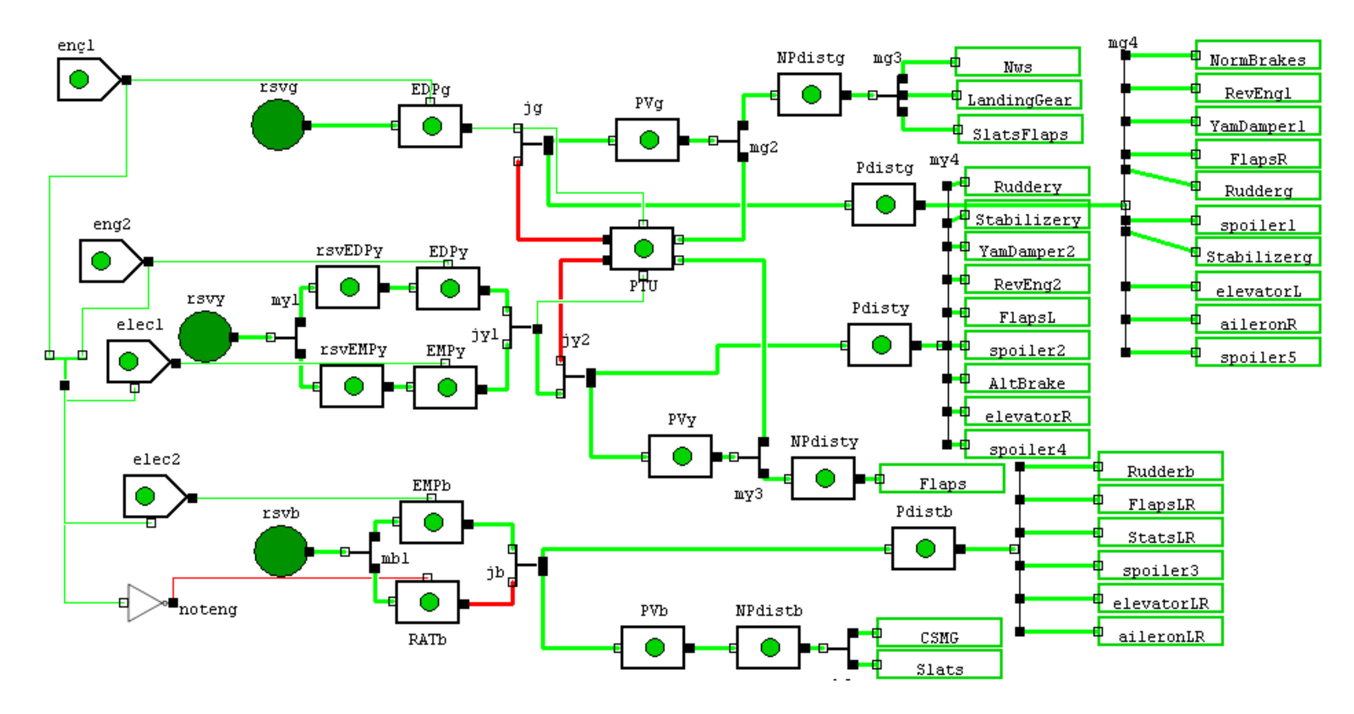


Fig. 7.12 – Modélisation du système Hydraulique avec OCAS

7.3.2 Validation du modèle

Pour faciliter les analyses sur ce modèle, nous ajoutons des composants supplémentaires pour l'observation des situations redoutées.

Ces composant « observateurs » permettent lors de l'évolution du système d'indiquer à tout moment si oui ou non l'état du système correspond à la situation redoutée. Dans notre cas, nous souhaitons observer la perte totale de puissance hydraulique. Pour exprimer ce comportement en AltaRica, nous créons un composant possédant autant d'entrées qu'il y a de lignes hydrauliques à surveiller et la valeur de ses entrées est représentée par un booléen exprimant la présence ou non d'alimentation. Ce composant doit signaler la perte totale, il possède donc un flux de sortie exprimant par un booléen si oui ou non le système se trouve dans cette situation. Le composant « observateur » n'est pas sujet aux défaillances. Il ne possède donc pas de variable d'état ni d'événements.

Le code complet des observateurs est le suivant :

```
node Obs_Total_Loss
 flow
  I1 : bool : in ;
  I2 : bool : in ;
  I3 : bool : in ;
  O : bool : out ;
assert
O = not(I1 or I2 or I3);
edon

node Obs_Dual_Loss
 flow
  I1 : bool : in ;
  I2 : bool : in ;
  O : bool : out ;
assert
O = not(I1 or I2);
edon
```

Ces noeuds AltaRica sont utilisés pour construire les 8 observateurs suivants :
- Perte Totale de l'hydraulique prioritaire (observateur `Phyd`) et non-prioritaire (observateur `NPhyd`),
- Perte de deux lignes hydrauliques : lignes Bleues et Jaunes prioritaire (observateur `GBPhyd`) et non-prioritaire (observateur `GBNPhyd`), Bleue et Verte prioritaire (observateur `YBPhyd`) et non-prioritaire (observateur `YBNPhyd`), Jaune et Verte prioritaire (observateur `GYPhyd`) et non-prioritaire (observateur `GYNPhyd`).

De plus, nous utilisons les sorties des noeuds `Pdistb`, `NPdistb`, `Pdisty`, `NPdisty`, `Pdistg`, `NPdistg` pour observer les pertes des lignes hydrauliques bleue (prioritaire et non-prioritaire), jaune (prioritaire et non-prioritaire) et verte (prioritaire et non-prioritaire).

La figure 7.13 montre une configuration du modèle du système hydraulique dans laquelle la situation redoutée perte des lignes bleues et jaunes non-prioritaires est atteinte. On voit que l'observateur `YBNPhyd` est dessiné en rouge alors que tous les autres observateurs sont dessinés en vert car les situations redoutées correspondantes ne sont pas atteintes.

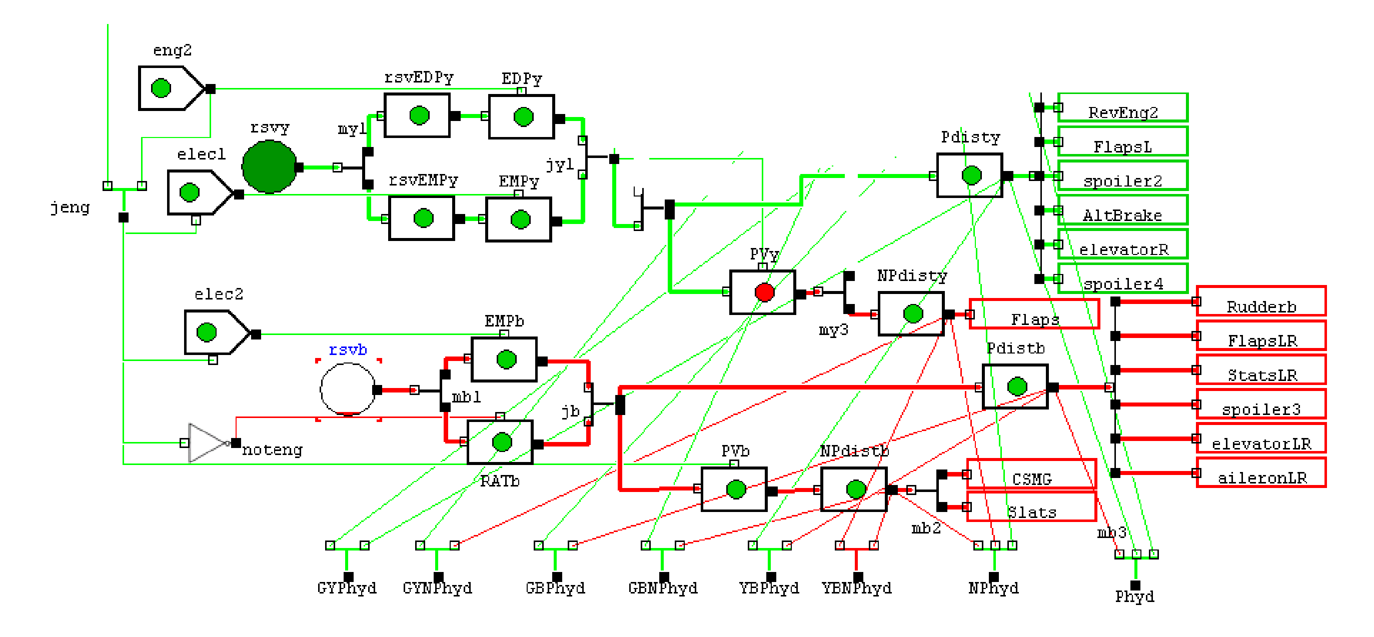


Fig. 7.13 – Observateurs associés aux situations redoutées

Nous utilisons la technique de *génération des scénarios* afin de vérifier les exigences associées au système hydraulique.

Pour chaque observateur nous générons les séquences associées et vérifions qu'il n'existe pas de séquences de taille plus petite que ce qui est stipulé par l'exigence de sûreté de fonctionnement qui correspond à cette situation redoutée. Puis nous utilisons Aralia[3] pour effectuer le calcul de probabilité et vérifions que les probabilités calculées sont acceptables.

Pour cette première version du modèle hydraulique nous nous sommes rapidement rendu compte que le système ne respectait pas les exigences de sécurité. En effet, il existe des pannes simples qui mènent à la perte des lignes jaune et verte, et des pannes doubles qui mènent à la perte totale d'hydraulique.

Un scénario qui mène à la perte des lignes jaune et verte est :

```
AltBrake.fail_leakage ; rsvy.update ; rsvy.update ; PTU.update ; rsvg.update ;
rvg.update ; PTU.update
```

Ce scénario ne comprend qu'une seule défaillance, celle du consommateur de la ligne jaune nommé `AltBrake` puis des événements `update` qui représentent des reconfigurations du système. Observons étape par étape ce scénario pour déterminer les corrections à apporter au système pour qu'il puisse tenir les exigences de sûreté de fonctionnement.

La figure 7.14 montre le système hydraulique après l'occurrence d'une fuite au niveau du consommateur `AltBrake`, ce composant est dessiné en rouge pour indiquer qu'il ne reçoit plus de puissance hydraulique.

[3] http ://www.arboost.com/aralia-page.htm

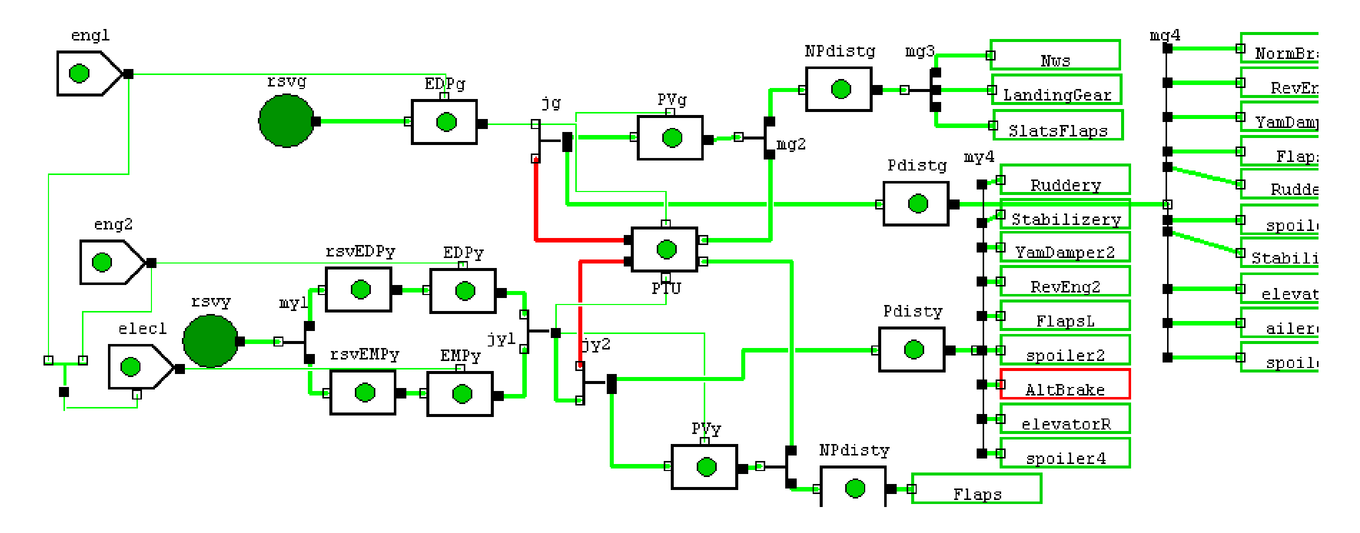


Fig. 7.14 – Scénario de perte des lignes Jaune et Verte - étape 1

La figure 7.15 montre l'effet sur le réservoir de la ligne jaune de la fuite du consommateur `AltBrake`. Après l'événement `rsvy.update`, le réservoir commence à baisser mais comme il n'est pas totalement vide les consommateurs de la ligne jaune sont toujours alimentés en puissance hydraulique.

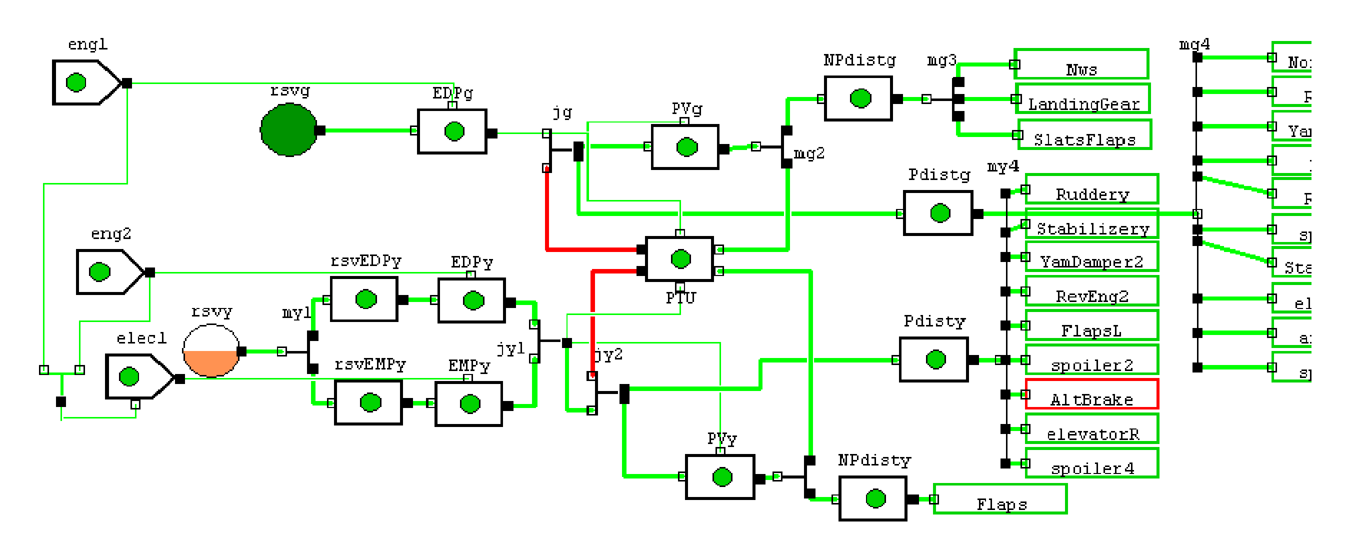


Fig. 7.15 – Scénario de perte des lignes Jaune et Verte - étape 2

La figure 7.16 montre l'effet sur le réservoir de la ligne jaune de la fuite du consommateur `AltBrake`. Après un second événement `rsvy.update`, le réservoir est vide et les consommateurs de la ligne jaune ne sont plus alimentés en puissance hydraulique.

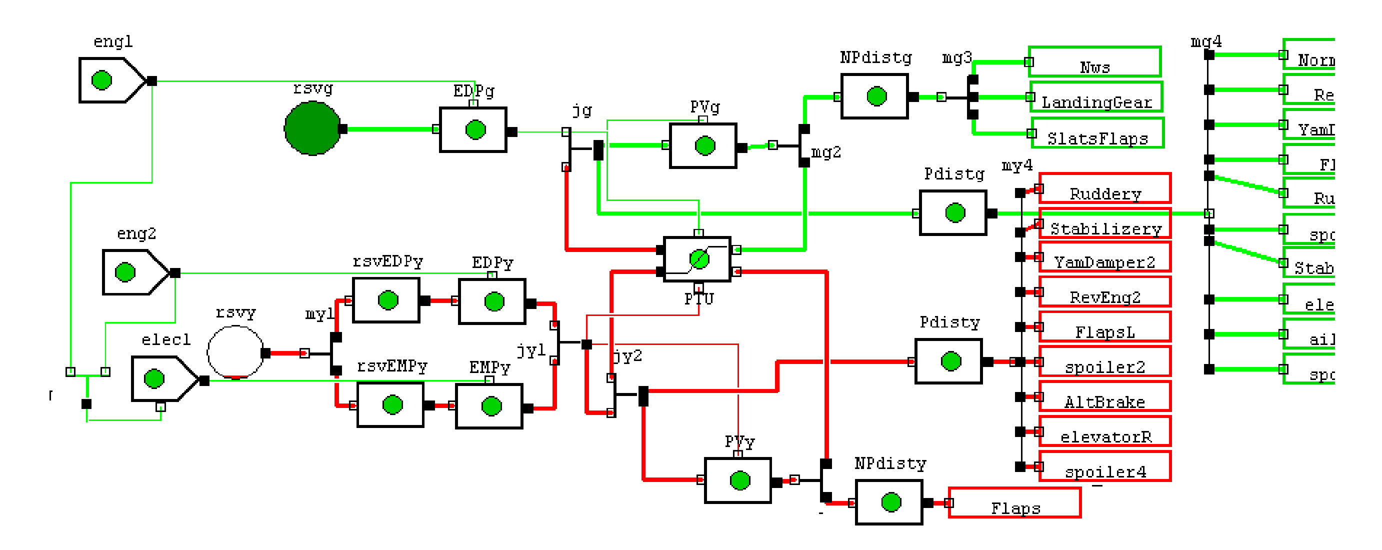


Fig. 7.16 – Scénario de perte des lignes Jaune et Verte - étape 3

La figure 7.17 montre l'effet de la reconfiguration du PTU, comme la ligne jaune n'est plus alimentée alors que la ligne verte l'est. Après l'événement `PTU.update`, le PTU alimente la ligne jaune avec la puissance hydraulique provenant de la ligne verte et les consommateurs de la ligne jaune ne sont à nouveau alimentés en puissance hydraulique.

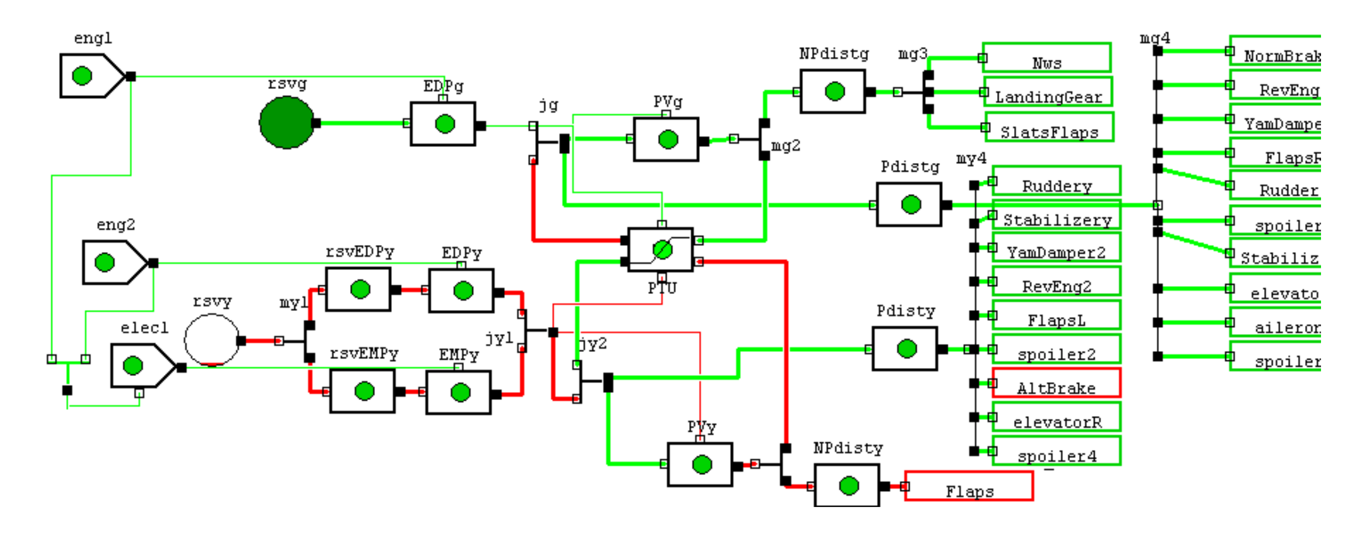


Fig. 7.17 – Scénario de perte des lignes Jaune et Verte - étape 4

La figure 7.18 montre l'effet sur le réservoir de la ligne verte de la fuite du consommateur `AltBrake`. En effet, la fuite se propage à travers le PTU de la ligne jaune vers la ligne verte après l'événement `rsvg.update` le réservoir de la ligne verte commence à baisser mais comme il n'est pas totalement vide les consommateurs des lignes jaune et verte sont toujours alimentés en puissance hydraulique.

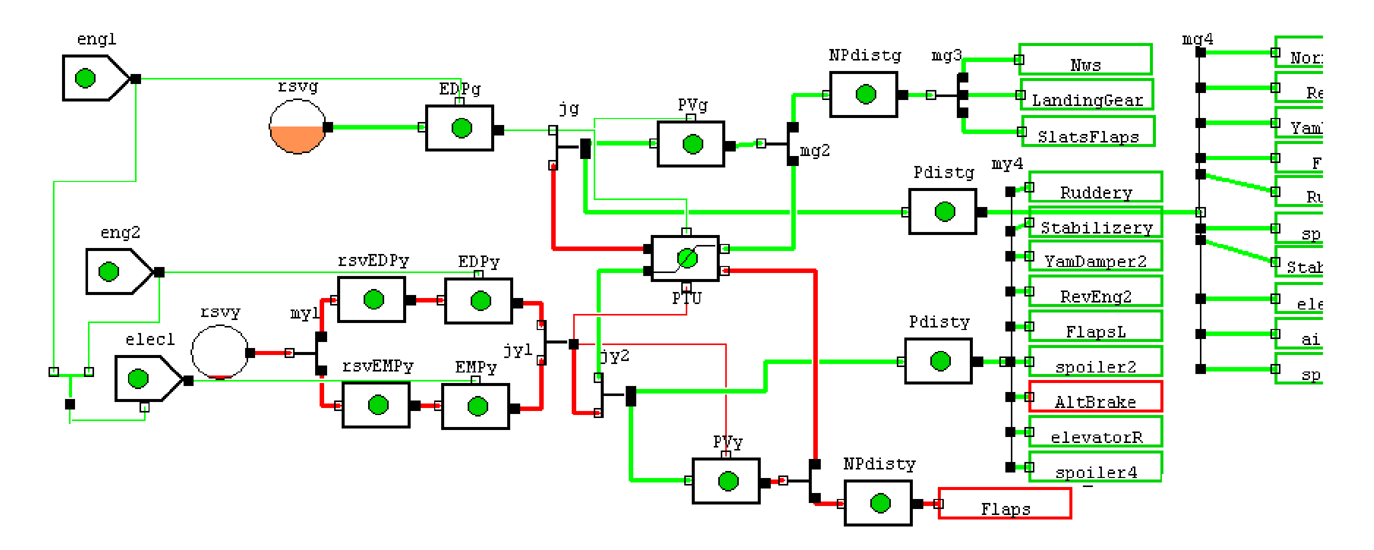


Fig. 7.18 – Scénario de perte des lignes Jaune et Verte - étape 5

La figure 7.19 montre l'effet final de la fuite du consommateur `AltBrake` : la perte des lignes verte et jaune. Après l'événement `rsvg.update` le réservoir de la ligne verte est vide donc les consommateurs des lignes jaune et verte ne sont plus alimentés en puissance hydraulique et après l'événement `PTU.update` le PTU se désactive mais cela n'a plus d'effet sur les consommateurs.

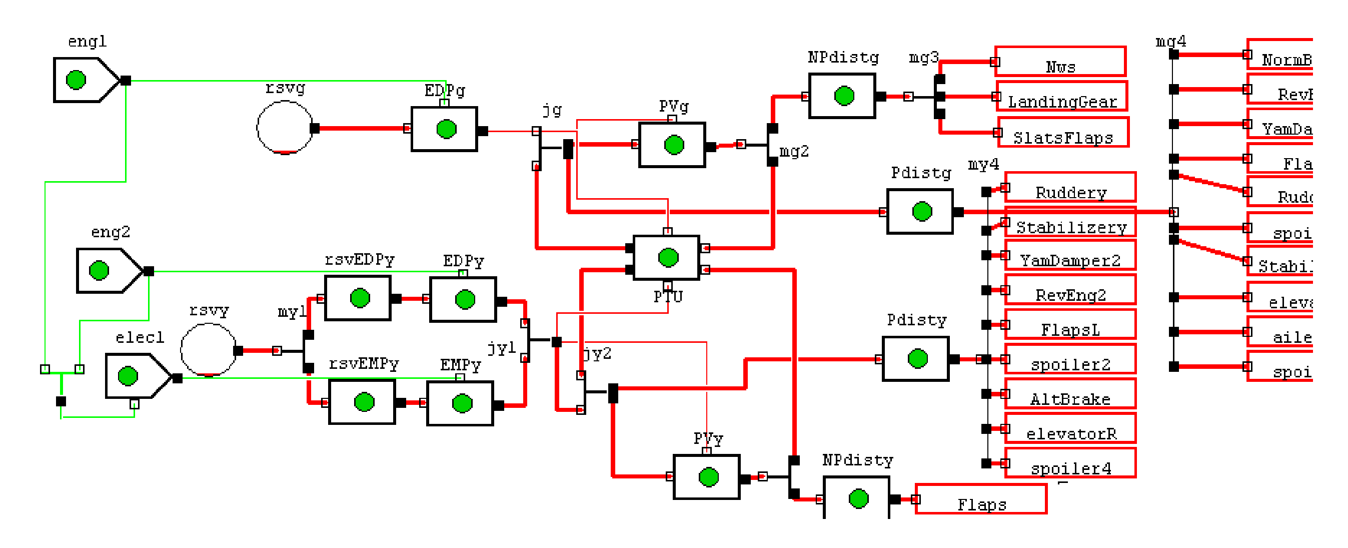


Fig. 7.19 – Scénario de perte des lignes Jaune et Verte - étape 6

Le problème réside dans la mauvaise activation du PTU lors d'une fuite d'une des deux lignes jaune ou verte. Il faudrait doter le PTU d'un capteur permettant de détecter les situations où l'absence de puissance hydraulique est due à une fuite et ne pas activer le PTU dans ce cas. Nous modifions le comportement du PTU : nous considérons qu'il ne propage plus la fuite sauf lorsque le capteur est défaillant. Nous ajoutons, au comportement du PTU, l'événement `fail_leakage` pour représenter la défaillance du capteur de fuite.

```
node PTU
 flow
  icone:[1,2]:private;
  I1^nom:bool:in;
  I1^rev:bool:out;
  O1^nom:bool:out;
  O1^rev:bool:in;
  A1:bool:in;
  A2:bool:in;
```

```
   O2^nom:bool:out;
   O2^rev:bool:in;
   I2^nom:bool:in;
   I2^rev:bool:out;
  state
   S:{ok, lost, leak};
   P12_rev:bool;
   P12_nom:bool;
   P21_rev:bool;
   P21_nom:bool;
  event
   fail_loss,
   fail_leakage,
   update;
  trans
   (S=ok or S=leak) |- fail_loss -> S:=lost;
   S=ok  |- fail_leakage -> S:=leak;
   (S=ok or S=leak) |- update -> P21_nom := (((not A1) and A2) and I2^nom),
                           P12_nom := (((not A2) and A1) and I1^nom),
                           P21_rev := (((not A1) and A2) and O1^rev),
                           P12_rev := (((not A2) and A1) and O2^rev);
  assert
   O1^nom = ((S=ok or S=leak) and P21_nom);
   O2^nom = ((S=ok or S=leak) and P12_nom);
   I1^rev = (S=leak and P12_rev);
   I2^rev = (S=leak and P21_rev);
  init
   S := true,
   P12_rev := false,
   P12_nom := false,
   P21_rev := false,
   P21_nom := false;
 edon
```

Après cette modification nous obtenons les résultats suivants :

Observateur	simple	double	triple	proba
Pdistb	12	0	1	4.8 e-4
NPdistb	13	1	0	5.8 e-4
Pdisty	1	222	159	1.0 e-4
NPdisty	13	23	0	4.0 e-4
Pdistg	1	232	141	1.0 e-4
NPdistg	3	218	135	3.0 e-4
YBPhyd	0	12	2663	4.8 e-8
YBNPhyd	0	170	286	2.4 e-7
GYPhyd	0	201	195	1.1 e-7
GYNPhyd	0	243	179	2.3 e-7
GBPhyd	0	12	2783	4.8 e-8
GBNPhyd	0	40	2834	1.8 e-7
Phyd	0	0	3146	5.1 e-11
NPhyd	0	1	2411	1.0 e-8

– La première colonne du tableau précédent indique le nom de l'observateur,

– la seconde colonne donne le nombre de pannes simples qui conduisent à la situation redoutée observée,
– la troisième colonne donne le nombre de pannes doubles,
– la quatrième donne le nombre de pannes triples et
– la dernière colonne indique la probabilité d'occurrence de cette situation.

Les probabilités sont calculées à l'aide de l'outil Aralia en considérant que toutes les défaillances représentées par un événement `fail_loss` ont un taux de défaillance de 10^{-4} par heure de vol et toutes les défaillances représentées par un événement `fail_leakage` ont un taux de défaillance de 10^{-5} par heure de vol.

Ces résultats sont satisfaisants car aucune panne simple ne conduit à la perte de deux ou trois lignes hydrauliques. Aucune panne double ne mène à la perte totale d'hydraulique prioritaire. En revanche il existe une panne double qui conduit à la perte totale de l'hydraulique non-prioritaire. En effet, lorsque les moteurs sont défaillants `eng1.fail_loss` et `eng2.fail_loss` alors les pompes moteurs EDPg et EDPy et les pompes électriques EMPy et EMPb ne sont pas alimentées et, par conséquent, les lignes jaune et verte sont perdues. Seule une pompe de la ligne bleue est alimentée par la "Ram Air Turbine" mais dans ce cas la valve de priorité PVb est fermée et seuls les consommateurs prioritaires de la ligne bleue sont alimentés. On peut supposer que ceci est acceptable puisque ces consommateurs sont considérés comme non-prioritaires. Il faudrait donc considérer que la perte des trois hydrauliques non-prioritaires est classée Hazardous et donc il serait acceptable qu'une panne double mène à cette situation. S'il n'est pas possible de changer la sévérité de cette situation alors l'analyse montre qu'il est impératif de modifier la logique de coupure de la vanne de priorité de la ligne bleue.

Toutes les probabilités calculées sont également acceptables, ce qui permet de fixer comme objectifs de développement pour les concepteurs des équipements du système hydraulique :
– le taux d'occurrence d'une perte d'équipement doit être inférieur à 10^{-4}par heure de vol et
– le taux d'occurrence d'une fuite de l'équipement doit être inférieur à 10^{-5}par heure de vol.

Bien entendu, il est possible qu'un équipement ne puisse pas tenir cet objectif général. Il est aisé de modifier le modèle AltaRica en tenant compte du taux de défaillance réel des équipements de façon à calculer à nouveau les taux de défaillances pour évaluer s'ils sont acceptables.

7.3.3 Modélisation géométrique

Le modèle géométrique correspond à une représentation en trois dimensions du système hydraulique. Cette représentation permet donc de visualiser le système tout en conservant certaines caractéristiques comme par exemple la taille des différents composants constituant le système ou la distance qui les sépare. Ces caractéristiques sont importantes lors des choix d'installation de ces composants dans l'avion. Par exemple, des exigences liées aux interactions électromagnétiques imposent une distance minimale entre équipements. Un autre exemple d'exigence recommande de ne pas installer de cables électriques en dessous de canalisations hydrauliques. Finalement, certaines exigences d'installation sont liées à la ségrégation entre équipements qui sont supposés tomber en panne de façon indépendante comme, par exemple, les lignes jaune, verte et bleue du système hydraulique. Tous ces types d'exigences peuvent être validées à l'aide des modèles géométriques.

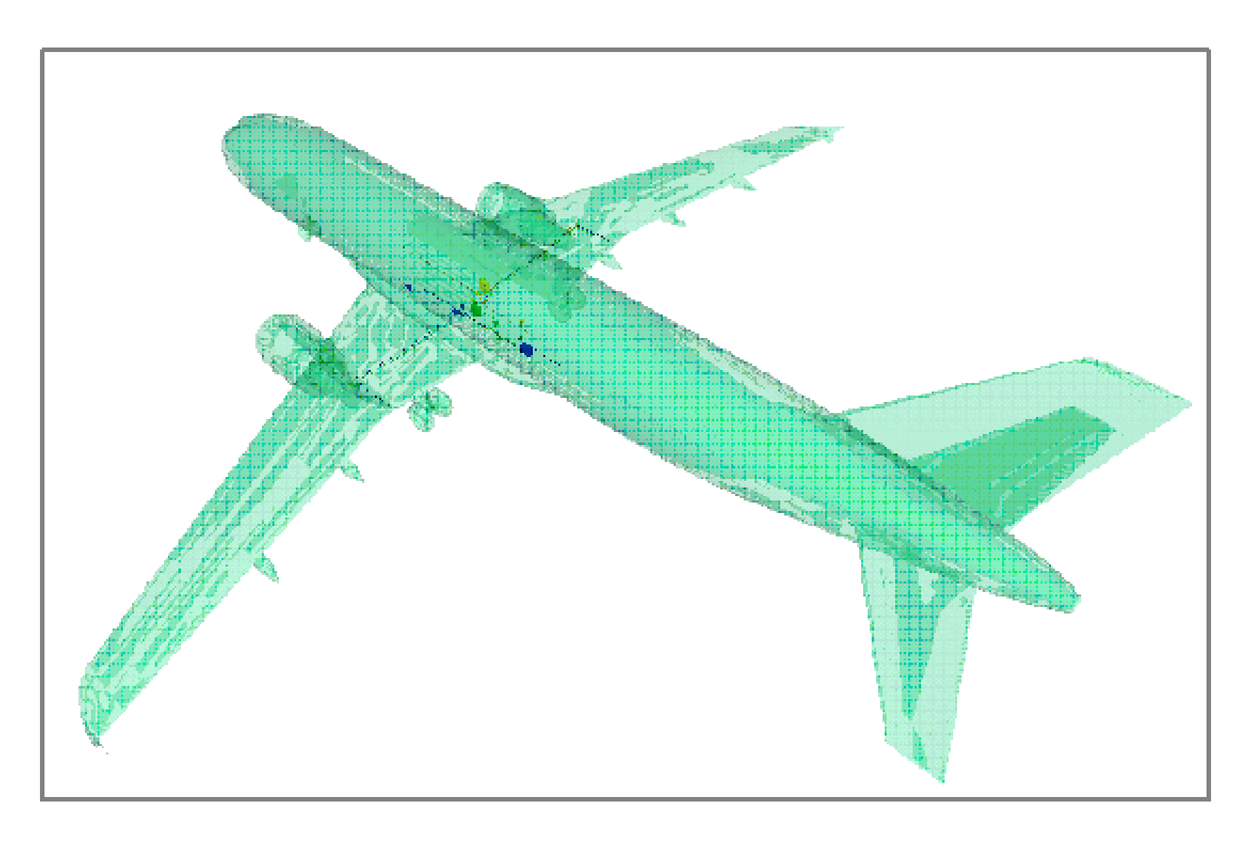


Fig. 7.20 – Modèle géométrique du système Hydraulique- vue générale

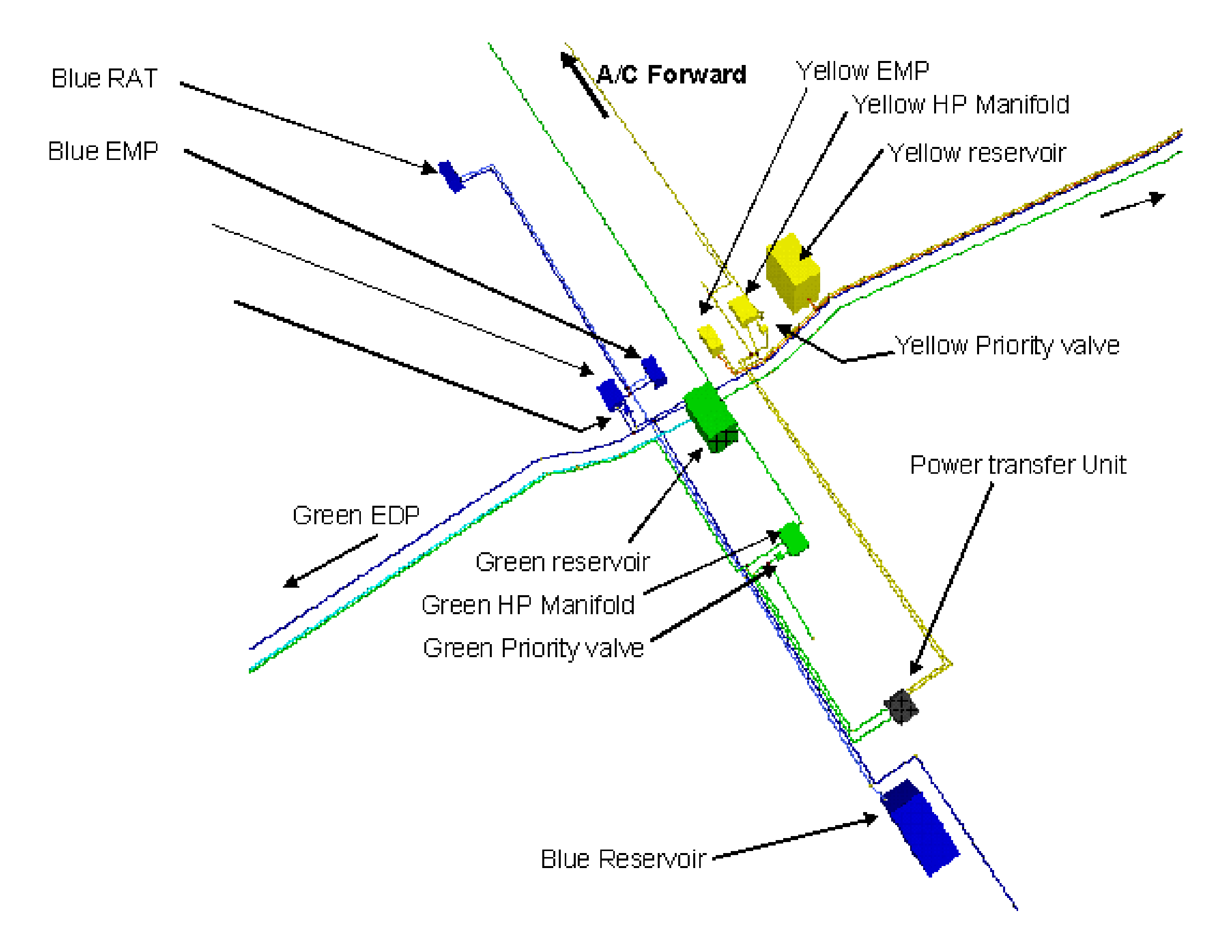


Fig. 7.21 – Représentation du système Hydraulique - vue détaillée

Une fois l'ensemble des équipements et des connexions modélisés, il est possible d'utiliser ce logiciel pour effectuer des analyses liées à la sûreté de fonctionnement. En effet, par des simulations d'éclatement de pneu et/ou de moteurs, le logiciel permet de vérifier si l'installation choisie est acceptable du point de vue des exigences de sûreté de fonctionnement.

Les modèles géométriques définis par IRIS permettent d'identifier les éléments qui sont impactés par les différentes projections de débris provenant de l'explosion simulée. En revanche, IRIS ou les outils similaires comme CATIA ne permettent pas de déterminer si une explosion donnée conduit à une des siutations redoutées qui ont été envisagées par les analystes de sûreté de fonctionnement.

Comme cela est visible dans la figure 7.22, l'interface du logiciel regroupe à la fois la visualisation du modèle en 3 dimensions (partie droite de la figure) et la visualisation de la liste des éléments impactés (« *Hitlist* ») suivant un angle donné (l'angle est représenté par la coupe de couleur marron dans la partie visualisation).

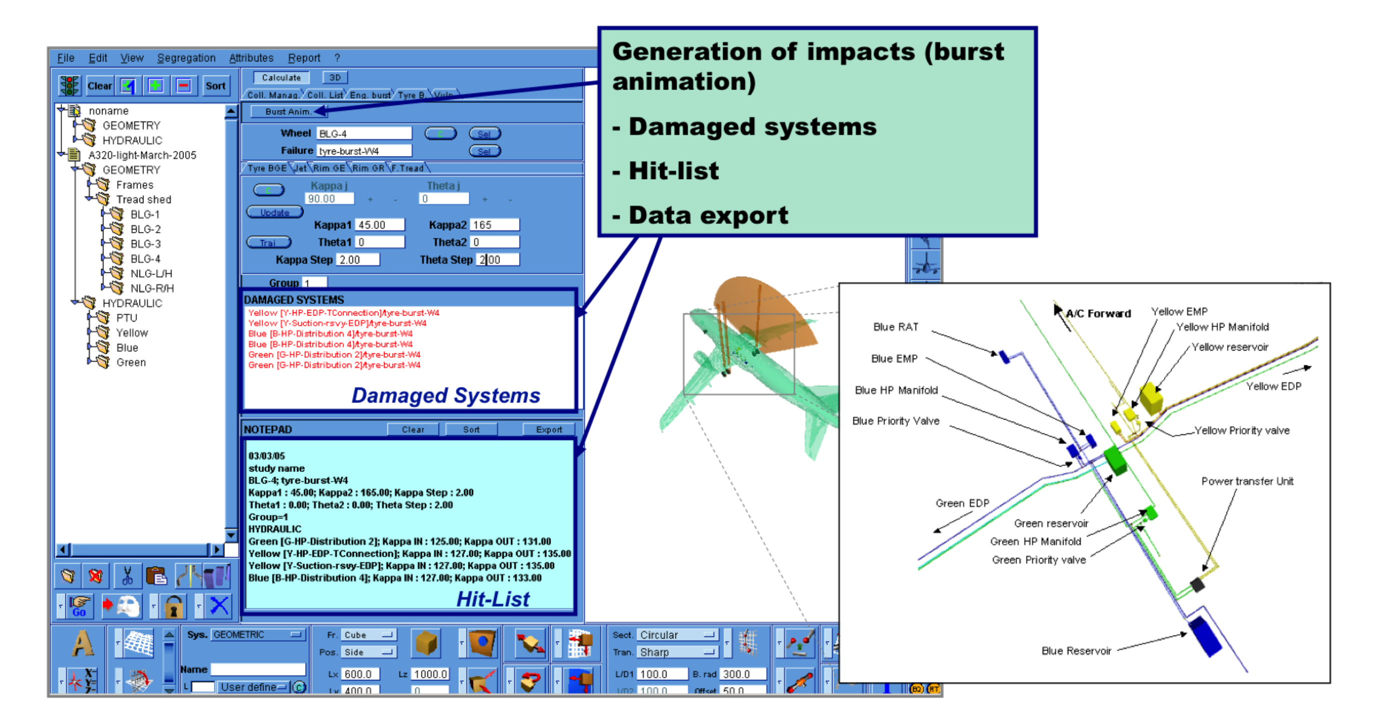


Fig. 7.22 – Représentation du système Hydraulique par IRIS

7.3.4 Les éléments impactés sous forme de *HitList*

Durant le projet ISAAC, nous avons travaillé en collaboration avec des équipes industrielles (Airbus, Alenia, Dassault Aviation) chargées de faire l'analyse des risques particuliers (éclatement pneu et éclatement moteur) sur des maquettes numériques. Chacune de ces équipes utilise des outils différents mais produisant le même type de résultats. Le résultat de ces analyses correspond à une liste des équipements impactés pour un certain éclatement.

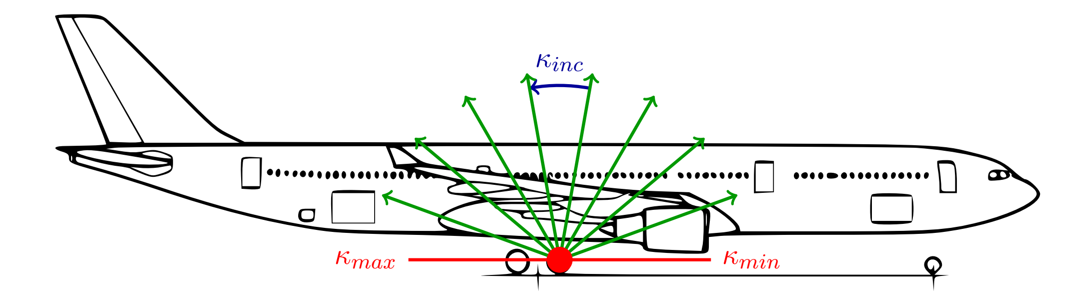


Fig. 7.23 – Analyse d'un éclatement pneu suivant l'angle κ

Comme vu sur la figure 7.23, lors d'une analyse d'un risque particulier comme un éclatement pneu, les débris suivent un angle κ qui varie entre κ_{min} et κ_{max}. Suivant les outils utilisés, lors d'une même analyse, il est possible de spécifier un angle θ qui couplé avec κ permet de représenter la zone visualisée comme un cône.

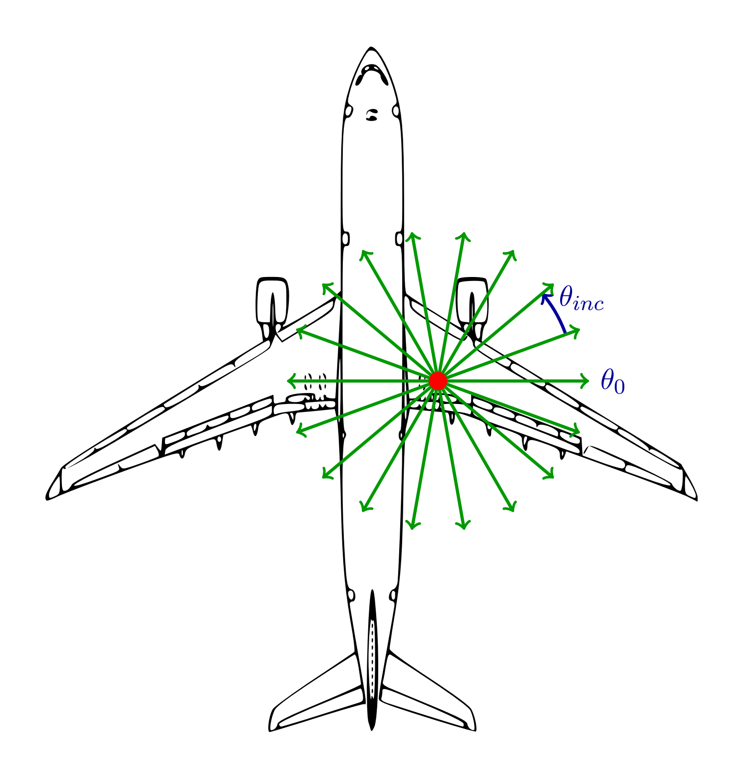


Fig. 7.24 – Analyse d'un éclatement pneu suivant l'angle *theta*

Pour une paire d'angles (θ, κ) donnée, ces logiciels fournissent la liste complète des équipements présents dans ce cône de débris. Les éléments intéressants dans ces analyses sont les composants impactés, mais aussi le degré d'impact de ces derniers. En effet, les équipements impactés possèdent des degrés différents d'impact suivant la distance qui les sépare de l'explosion. Certains seront complètement détruits alors que d'autres n'auront qu'une légère détérioration qui ne gênera pas son fonctionnement, mais qui empêchera de garantir un fonctionnement parfait de l'équipement.

Pour standardiser l'importation d'une liste d'éléments impactés produite par différents outils d'analyse, nous avons imposé le choix d'un formalisme. Une liste d'éléments impactés (*Hitlist*) doit être représentée par son nom et une liste des analyses effectuées. Chaque élément de cette liste (`hitlistitem`) doit correspondre à une étude d'un certain éclatement décrit par :

- le type d'analyse déterminé par exemple par le nom de l'élément qui va exploser (un éclatement d'un des pneus de l'avion ou d'un des moteurs),
- les équipements qui sont impactés ainsi que le degré de leur détérioration,
- les différentes trajectoires associées aux impacts

Prenons un exemple pour illustrer une *Hitlist* :

```
<hitlist name="Geometrical_with_IRIS"
         description="A320 tyre burst for hydraulic system">
...
<hitlistitem name="fail_tyre_burst_W4_Theta_minus15_9_Kappa_127_135" nb="10">
 <failmode component="G-HP-distribution-2" fm="impacted/>
 <failmode component="Y-Suction-rsvy-EDP" fm="impacted"/>
 <trajectorydetails theta="9.0" kappa_in="127" kappa_out="131"/>
 <trajectorydetails theta="11.0" kappa_in="127" kappa_out="131"/>
 <trajectorydetails theta="-15.0" kappa_in="129" kappa_out="135"/>
</hitlistitem>
...
</hitlist>
```

Dans cet exemple, nous pouvons déduire de l'analyse du fichier qu'il s'agit d'une *Hitlist* provenant du monde « `Geometrical_with_IRIS` » (information donnée par le champ `name`). Comme décrit dans la section 7.3.3, il s'agit d'une *Hitlist* provenant d'une analyse d'une maquette numérique en 3 dimensions réalisée avec le logiciel *IRIS*.

- Le champ `description` permet de noter qu'il s'agit ici d'une analyse d'éclatement pneu d'un avion de type Airbus A320 restreinte aux équipements du système hydraulique.
- Le champ `hitlistitem` regroupe des trajectoires qui impactent les mêmes équipements.
- L'exemple montre le groupe de trajectoires baptisé :
  `fail_tyre_burst_W4_Theta_minus15_9_Kappa_127_135`.
  Ces trajectoires sont liées à l'éclatement du $4^{\text{ième}}$ pneu pour des paires d'angle compris entre (-15°,135) et (9°,127°). Pour toutes ces trajectoires, les équipements `G-HP-distribution-2` (ligne distribution verte) et `Y-Suction-rsvy-EDP` (canalisation reliant le réservoir jaune avec la Pompe moteur jaune) sont détériorés.
- Le champ `trajectorydetails` est utilisé pour stocker des informations comme les angles des trajectoires qui conduisent à cet impact. Dans le cas illustré, la première ligne `trajectorydetails` indique que les trajectoires (9°, 127°), (9°, 129°) et (9°, 131°) impactent les mêmes équipements du système hydraulique. Les lignes suivantes concernent des trajectoires avec θ égal à 11° et à -15°.
- L'attribut `nb` indique le nombre de trajectoires regroupées dans ce `histlistitem` soit 10 trajectoires. Cet attribue est utile pour évaluer l'impact d'une allocation spatiale sur la tenue des exigences quantitatives de sûrete de fonctionnement.

7.4 Modélisation de l'allocation spatiale

Dans cette section, nous montrons l'intérêt d'une mise en relation des modèles AltaRica et des modèles géométriques IRIS par l'identification d'un scénario non prévu lié à l'installation. Nous montrons comment les analyses du nouveau modèle AltaRica (modèle incluant les placements des équipements dans l'avion) vont nous permettre d'améliorer une installation existante.

Nous illustrons également comment utiliser le MAPPINGMANAGER[4] pour mettre en relation les modèles AltaRica et IRIS à partir d'une *HitList*[5].

L'allocation spatiale que nous allons chercher à vérifier a été élaborée au cours de différentes réunions du projet ISAAC avec les correspondants AIRBUS.

[4]outil développé pendant la thèse présente page 137

[5]fichier correspondant aux résultats des analyses du modèle IRIS cf. section 7.3.4

Nous disposons de deux listes de composants pour le système hydraulique : l'une provenant du modèle IRIS et l'autre du modèle AltaRica. Ces deux listes ne se correspondent pas une à une, en effet les deux modèles ne décrivent pas avec le même niveau de détail certains aspects. Le modèle géométrique se focalise sur les équipements les plus encombrants comme les réservoirs ou les canalisations hydrauliques. Il décrira avec beaucoup de détail le cheminement du fluide depuis le réservoir jusqu'aux consommateurs. En revanche, certains mécanismes peu encombrants, comme la valve de priorité peuvent ne pas être décrits dans les modèles de pré-installation. Le modèle AltaRica se focalise sur les composants ayant une importance particulière du point de vue de la sûreté de fonctionnement soit parce qu'ils sont la source de problèmes (comme, par exemple, les fuites des consommateurs du système hydraulique) soit parce qu'ils contribuent à renforcer la sûreté du système (comme le composant *Power Transfer Unit*). Mais dans le modèle AltaRica du système hydraulique, nous avons choisi de représenter les différents « tuyaux » qui permettent le transport du fluide hydraulique, par un simple lien (connexion) entre les ports des noeuds AltaRica représentant les équipements reliés par ce tuyau. Les connexions entre les composants fonctionnels qui nous intéressent permettent de propager l'information de présence de fluide entre lesdits composants. Aucun comportement AltaRica et par conséquent aucune défaillance n'est associé à ces connexions. Or, la plupart des composants impactés par les risques particuliers sont ces tuyaux.

Nous avons fait le choix pessimiste d'associer l'endommagement d'un de ces tuyaux à la défaillance simultanée de tous les composants du modèle AltaRica qui lui sont connectés. Ainsi la perte du tuyau `G-Suction-rsvg-EDP` reliant le réservoir vert à la pompe verte est reliée avec la perte de `rsvg` et de `EDPg`. De même la perte du tuyau `Y-Suction-rsvg-EDP` reliant le réservoir vert à la pompe moteur jaune est reliée avec la perte de `rsvy`, de mais aussi de la pompe moteur `EMPy`.

L'allocation des composants du modèle AltaRica sur le modèle IRIS relie donc un ou plusieurs composants géométriques avec un ou plusieurs composants du modèle AltaRica. L'allocation choisie est décrite dans le tableau suivant :

Modèle IRIS	**Modèle AltaRica**
G-PV	PVg
G-ManifoldHP , G-HP-TConnection-HPManifold, G-HP-TConnection-PTU, G-HP-PTU-TConnection, G-HP-T-EDP-PTU, G-HP-EDP-TConnection, G-HP-Manifold-PV	EDPg, PVg, PTU
G-EDP	EDPg
G-Suction-rsvg-EDP	EDPg, rsvg
G-rsvg	rsvg
G-HP-NPriority-Distribution	NPdistg, Nws, LandingGear, SlatsFlaps
G-HP-T-PV-PTU-NPDistribution, G-HP-PV-TConnection	PTU, PVg, Npdistg
G-HP-Distribution1, G-HP-Distribution2, G-HP-Distribution3	Pdistg, NormBrakes, RevEng1,Yawdamper1, FlapsR, Rudderg, spoiler1, Stabilizer, elevator1, aileronR, spoiler5
Y-HP-PV-PTU	PVy, PTU
Y-PV	PVy
Y-ManifoldHP, Y-HP-PTU-HPManifold , Y-HP-T-EDP-EMP-PTU, Y-HP-TConnection, Y-Suction-T-rsvy-EMP-EDP, Y-Suction-rsvy-TConnection, Y-HP-HP-Manifold-T-PV, Y-HP-Manifold-T-PV, Y-HP-HP-Manifold-PV , Y-HP-T-EDP-EMP-PTU-HPManifold	EDPy, EMPy, PTU, PVy, Pdisty
Y-EDP	EDPy
Y-EMP	EMPy
Y-rsvy	rsvy
Y-Suction-rsvy-EDP, Y-Suction-rsvy-EMP, Y-HP-T-EMP-EDP, Y-HP-EMP-TConnection, Y-HP-EDP-TConnection	rsvy,EDPy, EMPy
Y-HP-T-NPDistribution	NPdisty
Y-HP-Distribution1, Y-HP-Distribution2, Y-HP-Distribution3	Pdisty, YawDamper2, RevEng2, FlapsL, spoiler2, AltBrake, elevatorR, spoiler4
B-PV	PVb
B-EMP	EMPb
B-Suction-rsvb-EMP, B-Suction-rsvb-RAT, B-HP-T-RAT-EMP, B-HP-RAT-TConnection, B-HP-EMP-TConnection, B-Suction-T-rsvb-RAT-EMP, B-Suction-rsvb-TConnection	rsvb, RATb, EMPb
B-RAT	RATb
B-rsvb	rsvb
B-HP-NPriority-Distribution	NPdistb, CSMG, Slats
B-ManifoldHP, B-HPManifold-PV,B-HP-TConnection-HPManifold	EMPb, RATb, PVb, Pdistb
B-HP-Distribution1, B-HP-Distribution2, B-HP-Distribution3	Pdistb
B-HP-Distribution4, B-HP-Distribution5	FlapsLR, SlatsLR, spoiler3, elevatorLR, aileronLR

Nous avons développé l'outil (MappingManager©) afin de faciliter la saisie de l'allocation des équipements du modèle AltaRica sur les équipements du modèle géométrique.

Le principe de fonctionnement de cet outil est simple (voir l'annexe A). Partant d'une liste de composants géométriques et d'une liste de composants fonctionnels issus d'un modèle AltaRica, l'outil permet la mise en correspondance des éléments par une interface graphique (cf. figure 7.25).

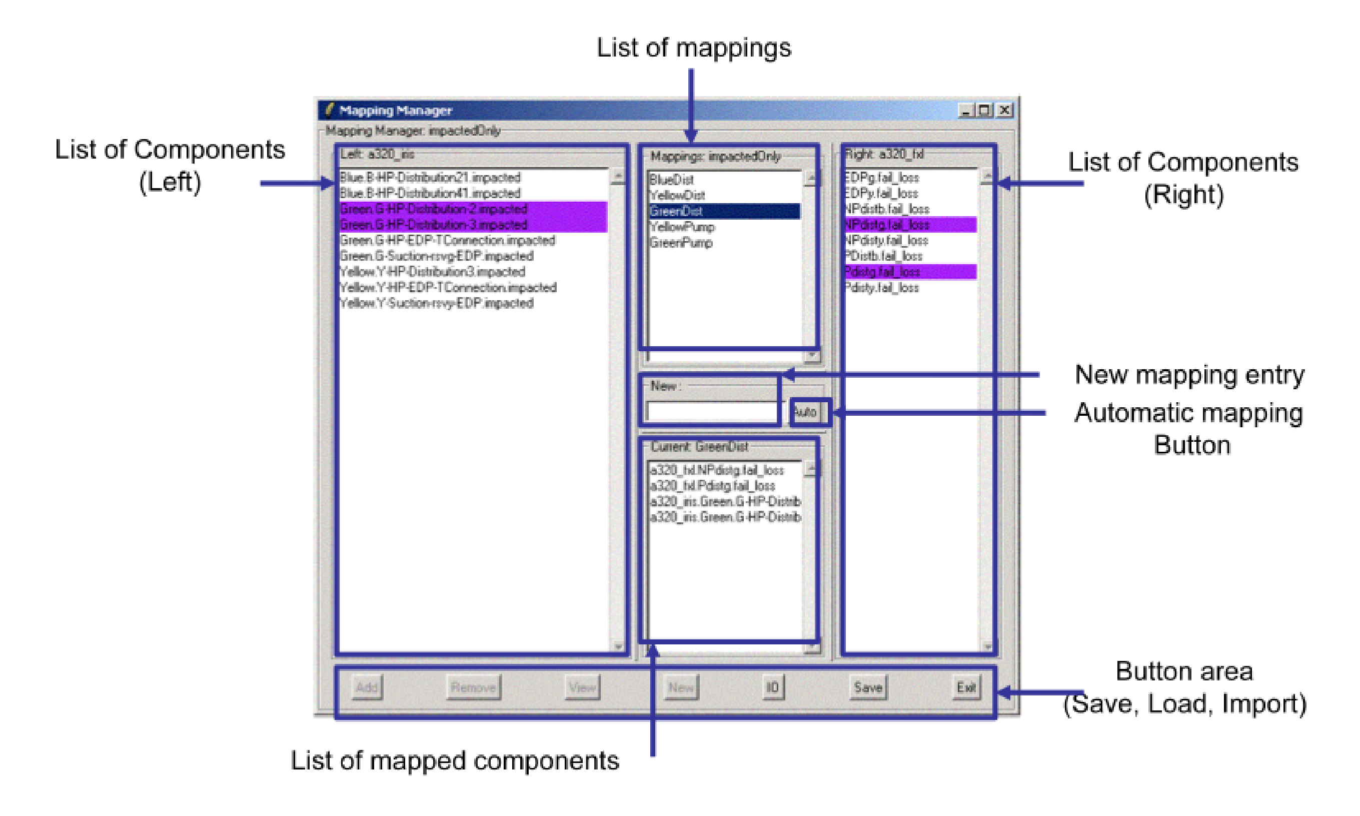


Fig. 7.25 – Interface du MappingManager

Exemple de liste de composants pouvant être utilisés par l'outil :

```
<failmodes name="Geometrical">
  ...
  <failmode component="G-HP-EDP-TConnection"
            fm="impacted"
            description="This T-connection links the Green Engine
                         Driven Pump with the High-Pressure Pipe"/>
  <failmode component="B-HP-Distribution-1"
            fm="impacted"
            description="Blue High Pressure Distribution line
                         (first part)"/>
  <failmode component="B-HP-Distribution-2"
            fm="impacted"
            description="Blue High Pressure Distribution line
                         (second part)"/>
  <failmode component="B-HP-Distribution-3"
            fm="impacted"
            description="Blue High Pressure Distribution line
                        (third part)"/>
  <failmode component="Y-EDP"
            fm="impacted"
            description="Yellow Engine Driven Pump"/>
  ...
</failmodes>
```

```
<failmodes name="Functional">
  ...
  <failmode component="EDPy" fm="fail_Loss"
            internal="MA320.Hyd.EDPy"
            description="Yellow Engine Driven Pump"/>
  <failmode component="Pdistb" fm="fail_leakage"
            internal="MA320.Hyd.Pdistb"
            description="Blue Priority distribution"/>
  ...
</failmodes>
```

Grâce à cet outil, il est possible de faire correspondre n éléments géométriques avec m éléments fonctionnels. Nous avons choisi d'utiliser le format « xml » pour représenter une allocation.

Dans l'exemple suivant, nous supposons que la liste des composants du monde géométrique est chargée dans la partie gauche de l'outil et elle est appelée « *Geometrical* ». De même, la liste des composants fonctionnels est chargée dans la partie droite et est appelée « *Functional* ». Ce fichier est constitué d'une série d'allocations (`map`). Chaque allocation comprend tous les noms des composants mis en relation. Pour chaque composant, plusieurs champs doivent être renseignés :

- le champ `component` donne le nom du composant impacté,
- le champ `fm` donne le nom de la défaillance qui est associée au composant,
- le champ `world` indique le monde d'où provient le composant. Les mondes possibles sont le monde fonctionnel (Functional) et le monde géométrique (Geometrical).

Tous les attributs optionnels trouvés dans la liste des composants, au moment de l'importation dans l'outil, sont ajoutés automatiquement dans le fichier.

```
<mapping name="A320 Hydraulic Light"
         left="Geometrical"
         right="Functional">
  ...
  <map name="Yellow Engine Driven Pump">
    <failmode world="Functional"
              component="EDPy"
              fm="fail_Loss"
              internal="MA320.Hyd.EDPy"
              description="Yellow Engine Driven Pump"/>
    <failmode world="Geometrical"
              component="Y-EDP"
              fm="impacted"
              description="Yellow Engine Driven Pump"/>
  </map>
  <map name="Blue Non Priority Distribution">
    <failmode world="Functional"
              component="NPdistb"
              fm="fail_loss"
              description="Blue Priority Distribution line">
   <failmode world="Functional"
              component="CSMG"
              fm="fail_leakage"
              description="CSMG consumer">
   <failmode world="Functional"
              component="Slats"
              fm="fail_leakage"
              description="Slat consumer">
    <failmode world="Geometrical"
              component="B-HP-distribution-1"
              fm="impacted" "
              description="Blue High Pressure Distrib line (part1)">
```

```
      <failmode world="Geometrical"
                component="B-HP-distribution-2"
                fm="impacted"
                description="Blue High Pressure Distrib line (part2)">
      <failmode world="Geometrical"
                component="B-HP-distribution-3"
                fm="impacted"
                description="Blue High Pressure Distrib line (part3)">
    </map>
    ...
  </mapping>
```

7.5 Vérification de l'allocation spatiale

Comme nous l'avons vu dans les chapitres précédents, nous représentons une allocation en introduisant dans un modèle AltaRica des défaillances de causes communes qui regroupent les défaillances des composants alloués à une même ressource. Dans ce chapitre, nous pourrions analyser de la même façon l'allocation spatiale en analysant l'effet sur le modèle AltaRica de la défaillance de chaque composant du modèle géométrique. Par exemple, la défaillance de l'équipement `G-Suction-rsvg-EDP` entraîne la perte des composants `EDPg` et `rsvg` ce qui conduit à la perte de la ligne verte jusqu'à ce que le PTU alimente les consommateurs verts avec l'hydraulique jaune.

Mais le modèle géométrique fournit à travers les *Hitlists* des informations supplémentaires sur les dépendances entre équipements que nous cherchons à exploiter. Nous introduisons des défaillances de cause commune qui regroupent les défaillances des composants du modèle AltaRica qui correspondent aux équipements impactés par une trajectoire de débris de pneu ou de moteur.

Pour enrichir le modèle AltaRica avec les informations provenant des *Hitlist*, nous utilisons l'allocation qui a été modélisée avec le (MappingManager©). L'analyse de l'allocation doit permettre pour chaque équipement impacté dans la *Hitlist* de trouver son ou ses correspondants dans le modèle AltaRica. L'outil (MappingManager©) génère automatiquement toutes les défaillances de causes communes qu'il faut ajouter au modèle AltaRica.

Une fois la mise en correspondance des 2 modèles établie, l'intégration des analyses d'éclatements consiste à ajouter des synchronisations sur les événements des composants AltaRica correspondants.

Pour chaque ligne `histlistitem` du fichier *Hitlist* $\mathcal{H}l$ et pour chaque équipement impacté, nous utilisons le fichier d'allocation géré par le MappingManager pour identifier le ou les composants correspondants aux composants géométriques. Par exemple le `histlistitem` intitulé `fail_tyre_burst_W4_Theta_minus15_9_Kappa_127_135` impacte les équipements `G-HP-distribution-2` et `Y-Suction-rsvy-EDP`. L'allocation présentée plus haut relie le composant `G-HP-distribution-2` avec les composants `Pdistg`, `NormBrakes`, `RevEng1`, `Yawdamper1`, `FlapsR`, `Rudderg`, `spoiler1`, `Stabilizer`, `elevator1`, `aileronR` et `spoiler5` et le composant `Y-Suction-rsvy-EDP` avec `rsvy`, `EDPy` et `EMPy`.

Cette méthode construit pour chaque `histlistitem` un vecteur de synchronisation $\overrightarrow{v}_{sync}$ des événements des composants AltaRica correspondants.

Par exemple, un nouvel événement global nommé `fail_tyre_burst_W4_Theta_minus15_9_Kappa_127_13` est ajouté qui regroupe les événements suivants :

- `Pdistg.fail_lost`,
- `NormBrakes.fail_leakage`,
- `RevEng1.fail_leakage`,
- `Yawdamper1.fail_leakage`,
- `FlapsR.fail_leakage`,
- `Rudderg.fail_leakage`,
- `spoiler1.fail_leakage`,
- `Stabilizer.fail_leakage`,
- `elevator1.fail_leakage`,

– `aileronR.fail_leakage`,
– `spoiler5.fail_leakage`,
– `rsvy.fail_lost`,
– `EDPy.fail_lost`
– `EMPy.fail_lost`.

Voici l'algorithme nous permettant de construire les synchronisations à ajouter au modèle AltaRica :

Algorithme 1 : Obtention des synchronisations

---

entrées : Une *Hitlist* $\mathcal{H}l = \{hl_1, hl_2, \ldots, hl_n\}$,
une allocation $\mathcal{M}app : \mathcal{C}_r \rightarrow \mathcal{C}_f$, où $\mathcal{C}_f$ est l'ensemble des composants du modèle AltaRica et $\mathcal{C}_r$ est l'ensemble des composants du modèle géométrique
sortie : Un vecteur de synchronisation $\overrightarrow{v}_{sync}$

pour $i \leftarrow 1$ **à** n **faire**
  $hl_i = \{cp_1, cp_2, \ldots, cp_m\}$
  pour $j \leftarrow 1$ **à** m **faire**
    si $\mathcal{M}app(cp_i) =$ « *vide* » **alors**
      retourner *« Pas de correspodance »*
    sinon
      $\overrightarrow{vi}_{sync} \leftarrow \mathcal{M}app(cp_j)$
  retourner $\overrightarrow{vi}_{sync}$

Nous obtenons donc dans le modèle AltaRica une nouvelle défaillance pour chaque analyse déduite de la *Hitlist*. Cette défaillance représente la synchronisation de tous les composants AltaRica impactés par une trajectoire donnée.

L'ajout de nouvelles défaillances dans le modèle AltaRica, ne doit pas invalider nos exigences de départ. Il faut donc réitérer nos analyses pour être sûr que de nouveaux scénarios ne violent pas nos exigences.

7.5.1 Nouvelles analyses

Nous effectuons les analyses des exigences du système hydraulique avec le modèle qui contient les défaillances de cause commune calculées par le MappingManager©. Puis nous comparons les résultats avec ceux obtenus avant la prise en compte de l'allocation spatiale de façon à déterminer si l'allocation est acceptable.

Nous avons étudié la hit-list comprenant les trajectoires associées aux éclatements des 4 pneus du train d'atterrissage central. Les résultats suivants ont été obtenus :

	avant allocation			après allocation		
Observateur	simple	double	proba	simple	double	proba
Pdistb	12	0	4.8 e-4	14	0	7.1 e-4
NPdistb	13	1	5.8 e-4	15	1	8.1 e-4
Pdisty	1	222	1.0 e-4	12	304	4.9 e-4
NPdisty	13	23	4.0 e-4	27	23	8.6 e-4
Pdistg	1	232	1.0 e-4	13	297	5.1 e-4
NPdistg	3	218	3.0 e-4	15	279	7.0 e-4
YBPhyd	0	12	4.8 e-8	0	168	3.5 e-7
YBNPhyd	0	170	2.4 e-7	0	406	7.1 e-7

```
GYPhyd      | 0      | 201    | 1.1 e-7 || 10     | 299    | 3.8 e-4
GYNPhyd     | 0      | 243    | 2.3 e-7 || 10     | 350    | 3.8 e-4
GBPhyd      | 0      | 12     | 4.8 e-8 || 0      | 182    | 3.6 e-7
GBNPhyd     | 0      | 40     | 1.8 e-7 || 0      | 226    | 5.8 e-7
------------+--------+--------+---------++--------+--------+---------
Phyd        | 0      | 0      | 5.1 e-11|| 0      | 151    | 3.2 e-7
NPhyd       | 0      | 1      | 1.0 e-8 || 0      | 140    | 2.7 e-7
---------------------------------------------------------------------
```

Afin de faciliter la comparaison, le tableau précédent reprend dans les 3 premières colonnes les résultats obtenus avant l'introduction des défaillances de mode commun, les trois dernières colonnes donnent les nouveaux résultats.

Les probabilités ont été calculées en gardant les mêmes taux de défaillances que précédemment pour les défaillances des composants. Pour les défaillances de mode commun, nous tenons compte du nombre de trajectoires représentées par une défaillance. La probabilité d'une défaillance de mode commun est égale au rapport du nombre de trajectoires représentées par cette défaillance et du nombre total de trajectoires considérées, multiplié par la probabilité d'éclatement d'un pneu . Dans le tableau précédent nous avons pris une probabilité d'éclatement de pneu égale à 10^{-4}. Nous avons considéré que 1024 trajectoires avaient été calculées par IRIS en faisant varier *theta* de -15° à 15° avec un incrément θ_{inc} de 2° et en faisant varier *kappa* de 53° à 179° avec un incrément κ_{inc} de 2°. Par conséquent, on obtient un taux de défaillance de $10 \div 1024 \times 10^{-4}$ (soit 0.9^{-6}) pour l'événement `fail_tyre_burst_W4_Theta_minus15_9_Kappa_127_135` qui regroupe 10 trajectoires.

Nous n'observons pas de dégradation importante des résultats (aussi bien qualitatifs que quantitatifs) pour les observateurs liés à la perte d'une seule ligne hydraulique. En revanche, en ce qui concerne la perte de deux lignes, les résultats ne sont pas acceptables. En effet, il existe dix événements simples qui conduisent à la perte des lignes jaune et verte (cf. les lignes du tableau précédent qui correspondent aux observateurs `GYPhyd` et `GYNPhyd`), ce qui viole l'exigence de sûreté de fonctionnement indiquant qu'une panne simple ne doit pas conduire à la perte de deux lignes hydrauliques. De plus, les probabilités associées sont supérieures au taux de défaillance requis : nous obtenons des probabilités de l'ordre de $3,8 \times 10^{-4}$ alors que le taux requis est de 10^{-5}.

Observons un scénario possible conduisant à la perte des lignes jaunes et vertes. Par exemple l'événement `fail_tyre_burst_W4_Theta_minus15_9_Kappa_127_135` déclenche les événements `Pdistg.fail_lost` qui conduit à la perte de la ligne verte prioritaire et `rsvy.fail_lost` qui conduit à la perte de puissance sur la ligne hydraulique jaune. On pourrait espérer que le PTU puisse alimenter la ligne jaune à partir de la ligne verte et permettre ainsi que la ligne jaune ne soit pas perdue. Mais l'événement global précédent déclenche également l'événement `FlapsR.fail_leakage` qui conduit au vidage progressif du réservoir vert et par conséquent à la perte de puissance de la ligne verte, ce qui aboutit à la perte des deux lignes.

Les résultats ne sont pas satisfaisants non plus pour la perte totale de l'hydraulique prioritaire (cf. la ligne du tableau précedent qui correspond à l'observateur `Phyd`) puisque 151 combinaisons de deux pannes conduisent à cette situation alors qu'il n'y en avait aucune avant l'allocation. De plus, les probabilités pour l'hydraulique prioritaire comme non prioritaire sont inacceptables puisque qu'elles sont très nettement supérieures à 10^{-9} (cf. les lignes du tableau précedent qui correspondent aux observateurs `Phyd` et `NPhyd`).

Pour corriger cette situation il est possible de modifier plusieurs aspects :

- En premier lieu, il est possible de modifier l'emplacement des équipements ou le routage des canalisations pour qu'ils ne soient plus impactés simultanément par une trajectoire. Par exemple, dans l'exemple précédent, il faut éloigner la ligne de distribution verte de la canalisation reliant le réservoir jaune aux pompes. Ceci est possible en faisant passer les canalisations de la ligne verte par l'arrière de l'aile et les canalisations de la ligne jaune par l'avant.
- Lorsqu'il n'est pas possible d'éloigner les équipements, il est possible de jouer sur les probabilités des nouveaux événements pour que les exigences quantitatives soient tenues. Par exemple, en

prenant une probabilité d'éclatement de pneu de 10^{-6}, les probabilités de la perte de deux lignes et de trois lignes deviennent acceptables. Ceci est possible sous l'hypothèse que la qualité du pneu est fortement supérieure à la qualité des pneus classiques. Cette approche a été considérée pour autoriser le vol du Concorde après l'accident de Gonesse.
- Il est également possible de revenir sur l'allocation des composants AltaRica sur le modèle géométrique. Comme nous l'avons dit précédemment, nous avons choisi d'être pessimiste en considérant que la perte d'un tuyau entraîne la perte de tous les composants qui y sont reliés. Il est possible de diminuer la conséquence des pertes des canalisations. Par exemple, nous avons considéré que l'endommagement de la canalisation reliant le réservoir et la pompe jaune entraîne la perte de la pompe mais également du réservoir et de l'autre pompe. On pourrait considérer que seule la pompe alimentée par cette canalisation est perdue et que le réservoir et l'autre pompe continuent à fournir de la puissance hydraulique. Dans ce cas, l'impact des trajectoires serait bien moindre que celui que nous avons calculé dans le tableau précédent.
- En dernier recours, il est possible d'ajouter des blindages de façon à protéger un équipement ou une canalisation contre des débris de pneu ou de moteur. Comme cette solution ajoute du poids à l'aéronef, elle est considérée comme peu satisfaisante.

7.6 Bilan

L'application que nous avons présentée dans ce chapitre nous a permis de montrer que les principes de modélisation et d'analyse de l'allocation d'une architecture fonctionnelle sur une architecture matérielle sont également valables pour étudier l'allocation spatiale d'équipements installés dans l'avion.

La collaboration avec les partenaires industriels et les laboratoires ayant travaillé dans le projet ISAAC a permis de s'assurer que l'approche proposée avait un intérêt industriel à assez court terme. L'intérêt du développement de l'outil MappingManager a été de convaincre les partenaires de la faisabilité de l'approche. L'outil a aussi permis de tester rapidement les formats d'entrée et de sortie négociés au sein d'ISAAC. Depuis la fin du projet ISAAC en janvier 2007, les principes de l'outil MappingManager ont été intégrés dans des outils industriels : sous la forme de macros Excel par Alenia (cet outil est utilisé sur des développements réels) et comme plug-ins dans l'outil CATIA par Airbus (cet outil est en cours de développement et n'a pas encore été déployé industriellement).

Nous aurions souhaité illustrer les techniques *de recherche d'allocation* à base de contraintes sur ce type d'exemple. Les modélisations géométriques fournies par les partenaires d'ISAAC nous semblent trop détaillées pour appliquer l'approche que nous proposons. Il serait bien entendu possible de poser le problème de collision entre les débris d'un pneu ou d'un moteur et des équipements sous la forme d'un problème de contraintes. Mais il s'agirait de domaines de contraintes (par exemple, des équations linéaires avec solutions réelles) qui s'éloignent des techniques présentées dans les chapitres 4 et 5. Il semble qu'il serait possible d'appliquer notre approche mais avec une modélisation géométrique plus abstraite que celle réalisée aujourd'hui à l'aide d'outils comme CATIA ou IRIS. Il s'agirait d'un modèle d'installation sous la forme d'un graphe des zones de l'avion et des routes interconnectant ces zones. Cette idée a été soumise à plusieurs partenaires industriels qui l'ont jugée intéressante mais nous n'avons pas pu obtenir d'informations précises sur les zones et les chemins de câbles qu'il serait pertinent d'utiliser dans un tel modèle.

Chapitre 8

Conclusion

8.1 Bilan des travaux effectués

Dans nos travaux, nous avons tout d'abord montré que le langage AltaRica présente une expressivité suffisante pour modéliser des systèmes basés sur des architectures fonctionnelles et physiques. Ce langage dispose d'un ensemble de concepts permettant la définition et l'analyse de l'allocation de fonctions sur des composants physiques. C'est notamment la construction `sync` de synchronisation qui permet aisément de définir une allocation dans un modèle AltaRica : la synchronisation permet de représenter l'impact des pannes matérielles sur les défaillances de fonctions supportées par les composants en panne .

De plus, la modélisation des `flow` représentant les connexions entre composants permet de représenter aisément la propagation soit d'une panne au niveau matériel, soit d'une défaillance au niveau fonctionnel.

Finalement, un modèle AltaRica peut supporter l'ensemble des informations nécessaires pour faire des analyses de sûreté de fonctionnement afin d'évaluer les exigences qualitatives et aussi pour quantifier les probabilités associées à différentes situations redoutées.

Nous nous sommes ensuite attachés à l'élaboration d'une méthode de génération d'allocation basée sur la résolution de contraintes, permettant de générer automatiquement, lorsqu'une solution existe, une allocation garantissant que l'ensemble de contraintes de départ est satisfait. Les contraintes exprimées portent sur les relations d'allocation, et permettent par exemple d'exprimer que deux fonctionnalités doivent être allouées sur deux ressources physiques distinctes (contrainte d'indépendance). La technique de génération d'allocation par résolution de contraintes peut proposer, lorsque plusieurs allocations vérifiant les contraintes existent, non pas une solution mais un ensemble de solutions. Il est ensuite du ressort d'un architecte du système d'analyser les différentes solutions et de choisir celle qui lui semble être optimisable selon ses propres critères.

La définition des contraintes d'allocation ne constituant pas une étape du processus actuel de développement décrit par l'ARP [47696], nous nous sommes ensuite intéressés à la déduction de ces contraintes d'allocation à partir de l'expression des exigences de sûreté de fonctionnement. Nous avons pour cela proposé une méthode d'interprétation des exigences de sûreté de fonctionnement en contraintes d'allocation, tenant compte à la fois des classes de criticité des exigences et des composants du systèmes concernés par les exigences.

Cette génération automatique de contraintes d'allocation depuis des exigences de sûreté de fonctionnement nous permet ensuite de définir une allocation satisfaisant les contraintes et par conséquent les exigences de sûreté de fonctionnement. Pour définir une allocation, nous procédons de manière itérative sur le nombre de ressources nécessaires pour atteindre la satisfaction de toutes les

contraintes. Nous optimisons ainsi le critère « nombre de ressources ».

Nous avons ensuite mis en oeuvre les méthodes proposées sur deux études de cas de systèmes avioniques réels.
- Nous avons appliqué la méthode intégrée de définition d'allocation satisfaisant des exigences de sécurité sur un système de Suivi de Terrain d'avion de chasse. L'allocation considérée ici est l'allocation de fonctions sur des ressources de calcul et de communication. Les exigences de sûreté de fonctionnement associées aux fonctions ont été interprétées en contraintes d'allocation, utilisées pour générer automatiquement (à l'aide d'un outil de résolution de contraintes), une ou plusieurs allocations possibles.
- Pour terminer, nous avons appliqué la méthode de vérification d'exigences de sûreté de fonctionnement sur un système défini dans l'espace en trois dimension, pour lequel l'allocation considérée est l'allocation de composants réels dans différentes zones d'un avion. Pour cela nous avons considéré l'ensemble des pannes ayant un impact géographique sur des zones avion (éclatement d'un pneu, éclatement moteur, etc.), et nous avons vérifié, pour chacune d'entre elles, que l'allocation des composants dans les zones garantissait l'absence de conséquence catastrophique.

8.2 Comparaison avec des travaux similaires

Le travail réalisé lors de cette thèse, c'est déroulé dans le cadre de deux projets de recherche industriels : le projet CARLIT sur les techniques de développement associées à l'avionique modulaire intégrée et le projet européen ISAAC sur les techniques d'analyse de la sécurité des systèmes aéronautiques. Dans ces deux projets, les choix techniques avaient été réalisé dès leur lancement. En particulier la langage de modélisation AltaRica avait été sélectionné. C'est pour cette raison que nous n'avons pas procédé à un état de l'art préalable. En revanche, nous avons suivis les travaux sur des sujets similaires de façon à pouvoir placer et comparer notre approche. Dans la suite de ce chapitre nous décrivons quelques travaux qui nous ont semblé particulièrement pertinents.

8.2.1 Analyses de modèles

AADL

Beaucoup de systèmes avioniques critiques sont aujourd'hui conçus à l'aide de langages de type ADL (Architecture Description Language). Dans un but de standardisation et de normalisation de la conception de systèmes avioniques, la communauté avionique a défini un langage commun, *AADL* [1] qui demeure en perpétuelle évolution. Ce langage est constitué d'une multitude de concepts de modélisation, mais ces concepts ne sont pas organisés de manière à décrire des modèles formels. En effet, l'interprétation des différents concepts n'étant pas formalisée, la seule solution actuelle, pour valider des modèles de ce langage consiste à les transformer dans un langage formel. C'est une fois que la description AADL est traduite dans un langage formel que des analyses de sûreté de fonctionnement peuvent être effectuées. Par exemple, des modèles décrits avec AADL peuvent être traduits en réseaux de Petri à l'aide de règles de transformation.

Dans ce contexte, les travaux d'Ana-Elena Rugina [RKK06, Rug05] se basent sur le langage source AADL étendu avec l'annexe *AADL Error Model Annex* [SA06]. Cette annexe permet de décrire des modèles d'erreur pour la modélisation du comportement en présence de fautes. La démarche générale proposée dans ces travaux est représentée dans la figure 8.1 : elle consiste à créer un modèle d'erreur du comportement de chaque composant en présence de ses propres fautes et éventuellement de ses réparations (sans tenir compte de son environnement). Des attributs permettent :
- d'ajouter aux événements une probabilité d'occurrence, ainsi que
- de définir des flux dédiés au déclenchement de transitions

[1] *Avionics Architecture Description Language*

ce qui permet de modéliser les aspects probabilistes et structurels associés à la propagation de panne.

Ensuite par un ensemble de règles de transformation itératives, l'auteur construit le modèle de réseaux de Petri pouvant être ensuite analysé [BAK04, KB96, BMT99].

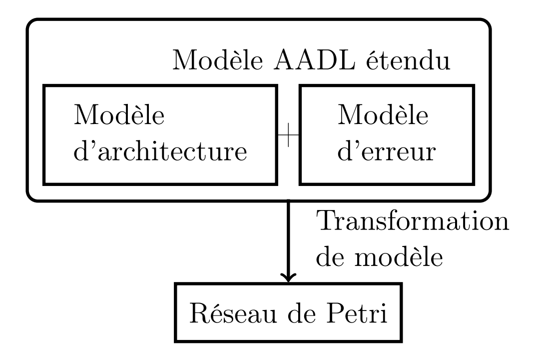


Fig. 8.1 – Transformation de modèles ADDL avec un modèle d'erreur en réseaux de Petri

La description du modèle d'erreur utilisé pour effectuer les analyses de sûreté de fonctionnement est intégralement exprimable à l'aide du langage AltaRica. Alors que la conception de modèle AltaRica est auto-suffisante pour effectuer ces mêmes analyses, les modèles AADL doivent être enrichis par ces modèles d'erreur. De plus l'auteur met en avant la nécessité de construire son modèle à base d'itération. En effet, la construction d'un réseau de Petri représentant un modèle *AADL* et un modèle d'erreur nécessite plusieurs parcours successifs des modèles d'entrée.

Ces itérations ne sont pas nécessaires dans le cas de la transformation d'un modèle AADL en langage AltaRica, comme cela est défini dans [DPS+08]. Plus précisément, dans le cas de la transformation d'un modèle AADL en un réseau de Pétri, une première itération est nécessaire sur le modèle AADL pour modéliser les comportements des composants du système en présence uniquement de leurs propres fautes et événements de réparation. Par conséquent, les composants sont modélisés comme s'ils étaient « isolés » du reste du système. Ensuite par des itérations successives, le modèle est complété en introduisant les dépendances entre les composants. Le modèle AADL de sûreté de fonctionnement est mis à jour à chaque itération en prenant en compte les nouvelles dépendances identifiées. Une fois le modèle AADL de sûreté de fonctionnement global crée, il est ensuite traduit en RdPSG (réseaux de Petri stochastiques généralisés) pour y vérifier des propriétés. La différence avec notre approche est dans le choix du langage. En effet, le langage AltaRica contient l'ensemble des composants constituant le système lui aussi directement intégré dans le modèle. Le comportement « individuel » des composants et le comportement global du système sont statiques, c'est-à-dire qu'ils sont directement décris dans le modèle et ne nécessitent aucune interprétation par différentes itération sur le modèle.

Dans sa thèse, Thomas Vergnaud [Ver06] présente une manière différente de traduction de modèle AADL en réseaux de Petri. Il faut noter que son objectif n'est pas l'analyse de sûreté de fonctionnement. L'auteur se base cette fois sur une notion définie directement par le langage AADL pour exprimer le comportement des composants. En effet, l'utilisation du langage AADL autorise la notion de *modes* de composant. Les modes permettent de modéliser la reconfiguration d'un composant en fonction d'événements pouvant être définis de façon locale. Ces événements permettent de déclencher un changement de mode pour représenter des architectures dynamiques de part l'évolution de leurs configurations. Cette notion est aussi très proche de la notion de *transition* en AltaRica mais comme il n'existe toujours pas de moyen pour exploiter ces modèles, l'auteur présente une technique de transformation des modèles AADL utilisant la notion de *modes* en réseaux de Petri pour les analyser.

Dans [MLMEP+05], les auteurs utilisent le langage de modélisation *MOC FTDF* (Model of Computation Fault Tolerant Data Flow), à partir desquels il est possible d'effectuer des analyses de performances et de sûreté de fonctionnement. Ces dernières se basent sur la synthèse d'arbres de défaillance à partir du modèle. Ce langage est aujourd'hui principalement employé pour la

modélisation de systèmes automobiles. La principale restriction de cette méthode est qu'elle ne s'applique qu'à des modèles statiques (par exemple, la modélisation des communications entre les ressources est unidirectionnelle) et pour exploiter les arbres produits, un certain nombre d'abstractions est nécessaire. Or, le langage AltaRica permet à la fois de modéliser les systèmes statiques, et de modéliser des systèmes dynamiques lorsque leur comportement devient trop complexe. En effet, les différents outils d'exploitation de modèles AltaRica permettent d'extraire des coupes de défaillances menant à une situation observée à partir de modèle statiques et dynamiques.

8.2.2 Définition d'allocation par résolution de contraintes

Dans [HCDJ06, CHD+04, HCDJ08], les auteurs présentent une approche, basée sur la programmation par contraintes, permettant de guider la définition d'une architecte de système temps-réel dans l'allocation de tâches périodiques sur des process distribués.

Tandis que dans le cadre de nos travaux, les contraintes étudiées sont du type « deux tâches doivent être indépendantes », qui sont traduites en contraintes d'allocation du type « indépendance entre les ressources supportant les tâches », dans l'approche proposée par les auteurs les contraintes expriment des temps de réponses à garantir sur l'exécution de tâches périodiques et sont traduites en contraintes d'ordonnancement portant sur les priorités des process exécutant les tâches, leurs périodicités, etc.

Finalement, l'allocation représente l'architecture temps-réel et donc la répartition des tâches sur différents process ayant différentes priorités et périodicité. Cette allocation doit satisfaire les contraintes des temps de réponse associés aux différentes tâches.

8.2.3 Conclusion

En conclusion, il nous semble que certains concepts d'AltaRica sont très adaptés au problème de la modélisation du problème d'allocation. Tout d'abord, l'aspect formel du langage permet d'effectuer des analyses aussi bien qualitatives que quantitatives sans qu'il soit nécessaire de traduire les modèles dans un autre langage. Le possibilité de construire des modèles compositionnels ainsi que la notion de synchronisation d'évènements permettent de construire et d'analyser efficacement un grand nombre de possibilités d'allocations. En effet, il n'est pas nécessaire de modifier la modélisation des architectures fonctionnelles ou matérielles pour traiter une nouvelle allocation. Seule la relation de synchronisation est à modifier pour traiter une nouvelle allocation.

Nous n'avons pas retrouvé à ce jour de langage de modélisation offrant des avantages équivalents lorsque nous nous intéressons à la sûreté de fonctionnement. l'approche AADL représente l'alternative la plus séduisante, mais il reste encore à améliorer les outils supportant les analyses de sûreté de fonctionnement pour atteindre le niveau d'AltaRica. Un grand avantage d'AADL est qu'il permet de traiter d'autre point de vue que la sûreté de fonctionnement comme les performances temps réel. Ceci dit, AADL reste confiné à la modélisation des systèmes informatiques. Ce qui n'est pas satisfaisant lorsqu'on s'intéresse à des problèmes d'allocation de systèmes hétérogènes (électriques, hydrauliques, commandes de vol ...) sur les équipements de l'avion.

8.3 Perspectives

Dans cette section, nous dressons un bref aperçu d'un ensemble de perspectives que nous avons identifiées. Nous commençons par une perspective technique d'amélioration de notre méthode de résolution de contrainte. Nous présentons ensuite les perspectives d'exploitation de nos travaux avant de présenter une perspective générale de nos travaux.

8.3.1 Utiliser l'existant

Une première perspective de travail présenté consiste à intégrer l'approche dans le processus de développement existant. En particulier, les techniques de modélisation et de vérification de l'allocation permettent d'assister les analystes de type CCA[2] comme les analyses de risque particulier (éclatement pneu, moteur, ...).

Il serait également possible d'accélérer les analyses de l'impact sur la sûreté de fonctionnement des allocations de fonction sur les modules IMA et les réseaux partagés comme l'ADCN.

Les techniques de génération automatique de l'allocation devraient aider les concepteurs de systèmes et les concepteurs de la plateforme avionique à converger plus rapidement vers de solutions d'architecture d'allocation.

8.3.2 Encore plus loin

Les techniques de génération à base de contraintes ont été appliquées dans cette thèse en se focalisant sur le respect des exigences de sûreté de fonctionnement. Or, d'autres familles d'exigences peuvent être traitées de la même façon : Dans le cadre du projet CARLIT, des contraintes liées aux exigences de performance temps-réel (respect des échéances des traitements de communications), de compatibilité électromagnétique (taux de perturbation des messages émis sur le réseau) ont également été développées.

Une perspective de notre travail serait le développement d'un outil d'aide à la conception de systèmes avioniques qui permettrait de trouver les contraintes provenant des différents points de vue que l'on peut associer sur un système.

Une dernière perspective concerne l'allocation de systèmes dans un espace en trois dimensions décrivant un avion. De nouveaux types d'exigences peuvent être identifiés pour aider au choix d'une allocation spatiale non seulement sûre, mais aussi optimisée. Un critère d'optimisation intéressant serait celui de la facilité d'atteindre les composants de l'avion présentant le plus fort taux de pannes, et donc les plus sujets à des activités de maintenance.

Les activités de maintenance représentant un coût très élevé sur l'exploitation d'un avion de ligne, le critère de rapidité de maintenance est un critère non négligeable. Il pourrait être aisément pris en compte dans notre analyse en injectant des contraintes supplémentaires dans notre système de contraintes d'allocation afin que les composants les plus sujets aux pannes soient disposés dans les zones facilement accessibles de l'avion. Citons l'exemple d'un avion sur lequel une simple opération de maintenance d'applications logicielles nécessite le démontage de la structure avion, et notamment le démontage des ailes. Ce type d'allocation des composants dans l'avion présente un coût considérable à l'exploitant. Par conséquent, le critère de l'accessibilité des éléments susceptibles de nécessiter des activités de maintenance est un critère d'optimisation légitime dans le choix d'une allocation spatiale.

[2]Common Cause Analysis

Annexe A

Manuel d'utilisation du MappingManager ©

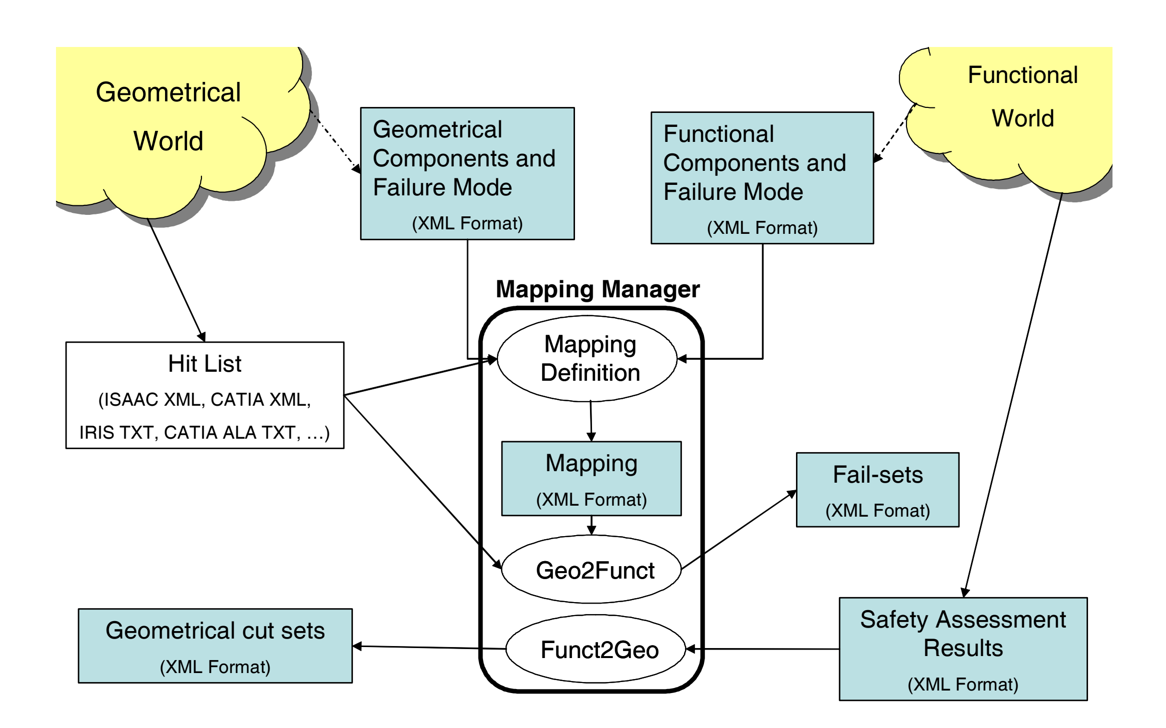
MapManager User Manual

August, 24th , 2006
ONERA

1 A tool for the management of Geometrical/Functional mappings

1.1 Goal of the tool

The goal of this tool is to manage mappings of components from the Geometrical world (CATIA, IRIS, …) with components of the functional world (StateMate, SCADE, …). The mapping is used to transform a list of impacted components from the Geometrical world into a group of simultaneously failed components in the functional world. Conversely, the tool can be used to transform safety assessment results obtained in the functional world as, for instance, a list of minimal cut sets into a list of impacted components from the geometrical world.



1.2 Formats

Formats for Hit Lists, Mapping files and Fail sets were defined as a result of ISAAC WP2 and were modified during ISAAC Paris meeting (Nov. 2005). Formats for Safety Assessment results and Geometrical cut sets were defined during ISAAC Turin meeting (March 2006).

In the following we make a distinction between:

- **compulsory** tag or attribute in bold font, this information is compulsory for the mapping manager to be able to create and store mappings and to generate failure sets.
- optional attribute in normal font, this information is not compulsory for the mapping manager but it could be necessary for the Geometrical or Functional tools. An optional attribute is not analysed by the mapping manager, so you can add attributes with any name you want as long as it is different from the names of the compulsory attributes. Optional attributes are stored in the mapping files and they are included in the failure set files generated.

Component and Failure Mode List

The proposed format for imported component list consist of a `failmode` item for each pair (component, associated failure mode) found in the Geometrical or Functional model. Attribute `component` is the name of the component and attribute `fm` is the name of the failure mode.
Other attributes were discussed during ISAAC Paris meeting, they are optional. The value of attribute `internal` is an internal name for the component that is not displayed in the mapping manager but that is useful for the functional world tools (especially for Prover implementation line). Attribute `description` can be used to include a textual description of the failure mode. Attribute `zone` can be used to store information about the zone where an equipment is installed.

Two examples of Component Lists:

```
<failmodes name="Geometrical">
  …
  <failmode component="G-HP-EDP-TConnection" fm="impacted"
description="This T-connection links the Green Engine Driven Pump with the
High-Pressure Pipe"/>
  <failmode component="B-HP-Distribution-1" fm="impacted" description="Blue
High Pressure Distribution line (first part)"/>
<failmode component="B-HP-Distribution-2" fm="impacted" description="Blue
High Pressure Distribution line (second part)"/>
<failmode component="B-HP-Distribution-3" fm="impacted" description="Blue
High Pressure Distribution line (third part)"/>
  <failmode component="Y-EDP" fm="impacted" description="Yellow Engine
Driven Pump"/>
  …
</failmodes>
```

```
<failmodes name="Functional">
  …
  <failmode component="EDPy" fm="fail_loss" internal="MA320.Hyd.EDPy"
description="Yellow Engine Driven Pump"/>
  <failmode component="Pdistb" fm="fail_leakage"
internal="MA320.Hyd.Pdistb" description="Blue Priority distribution"/>
  …
</failmodes>
```

Hit List

The tool deals with a generic format described in the following as well as with Proprietary formats (IRIS textual format, CATIA XML and textual formats).

A hit list contains several `hitlistitem` parts , each `hitlistitem` contains the list of impacted components. Futhermore, an hit list item may contain information related with the trajectories that caused it. The tag `trajectorydetails` is used to store information such as trajectory angles. The attributes included in a trajectorydetail line such as theta, kappa_in and kappa_out are optional. They are generally specific to a Geometric World implementation line : CATIA v5, IRIS or ICAD Disk Burst World.

```
<hitlist name="Geometrical" description="A320 tyre burst hit list for
hydraulic system">
…
 <hitlistitem name="fail_tyre_burst_W4_K127">
  <failmode component="B-HP-distribution-1" fm="impacted/>
  <failmode component="Y-EDP" fm="impacted"/>
```

```
  <trajectorydetails theta="5" kappa_in="131" kappa_out="145"/>
  <trajectorydetails theta="4" kappa_in="138" kappa_out="145"/>
  <trajectorydetails theta="3" kappa_in="142" kappa_out="145"/>
  <trajectorydetails theta="2" kappa_in="144" kappa_out="145"/>
 </hitlistitem>
…
</hitlist>
```

Mapping File

A mapping is a table that relates component and failure modes of the functional world with components and failure modes of the geometrical world. In the following example, we suppose that the list of Geometrical components was imported in the left side of the Mapping manager and it is called "Geometrical" whereas the other list was imported in the right side and is called "Functional". For each line of the table a `map` item is added, it contains all related components. For each component a `failmode` item is added, its attribute `world` is either equal to the name of the list of components from the Geometrical or from the Functional world, `component` is the name of the impacted component and `fm` is the failure mode associated to this component. All optional attributes found in the component lists are stored in the mapping.

```
<mapping name="A320 Hydraulic Light" left="Geometrical" right="Functional">
…
  <map name="Yellow Engine Driven Pump">
    <failmode world="Functional" component="EDPy" fm="fail_loss" internal=
"MA320.Hyd.EDPy" description="Yellow Engine Driven Pump"/>
    <failmode world="Geometrical" component="Y-EDP" fm="impacted"
description="Yellow Engine Driven Pump"/>
  </map>
  <map name="Blue Priority Distribution">
    <failmode world="Functional" component="Pdistb" fm="fail_leakage"
internal="MA320.Hyd.Pdistb" description="Blue Priority Distribution line">
    <failmode world="Geometrical" component="B-HP-distribution-1"
fm="impacted" " description="Blue High Pressure Distribution line (first part)">
    <failmode world="Geometrical" component="B-HP-distribution-2"
fm="impacted" " description="Blue High Pressure Distribution line (second part)">
    <failmode world="Geometrical" component="B-HP-distribution-3"
fm="impacted" " description="Blue High Pressure Distribution line (third part)">
  </map>
…
</mapping>
```

Fail Set

A fail set file contains several failure sets, each failure set is made of a list of failure modes and a list of trajectory descriptions.

```
<failsets name="Impact on Functional Model of tyre burst analysis"
failmodes="Functional">
…
 <failset name="a_fail_set_name">
  <failmode component="EDPy" fm="fail_loss" internal="MA320.Hyd.EDPg"
description="Yellow Engine Driven Pump">
  <failmode component="Pdistb" fm="fail_leakage">
   <trajectorydetails theta="5" kappa_in="131" kappa_out="145">
   <trajectorydetails theta="4" kappa_in="138" kappa_out="145">
   <trajectorydetails theta="3" kappa_in="142" kappa_out="145">
   <trajectorydetails theta="2" kappa_in="144" kappa_out="145">
```

```
 </failset>
…
</failsets>
```

Safety Assessment Results

A Safety Assessment result file contains several cutsets, each cutset is made of a list of failure modes or failure sets. The size of the cutset indicates the number of the failure modes or failure sets included in the cutset, notice that a failmode that belongs to a failset is not counted.

```
<cutsets failsets="Safety Results of the Hydraulic Functional Model
extended with tyre burst information" failmodes="Functional">
…
 <cutset size="2" name="a_cut_set_name">
  <failmode component="eng2" fm="fail_loss">
  <failset name="a_fail_set_name">
      <failmode component="EDPy" fm="fail_loss">
      <failmode component="Pdistb" fm="fail_leakage">
  </failset>
</cutset>
…
</cutsets>
```

Geometrical Cut Sets

A Geometrical cut set file is similar to a safety assessment file where names of functional components are replaced with names of geometrical components that are mapped to them.

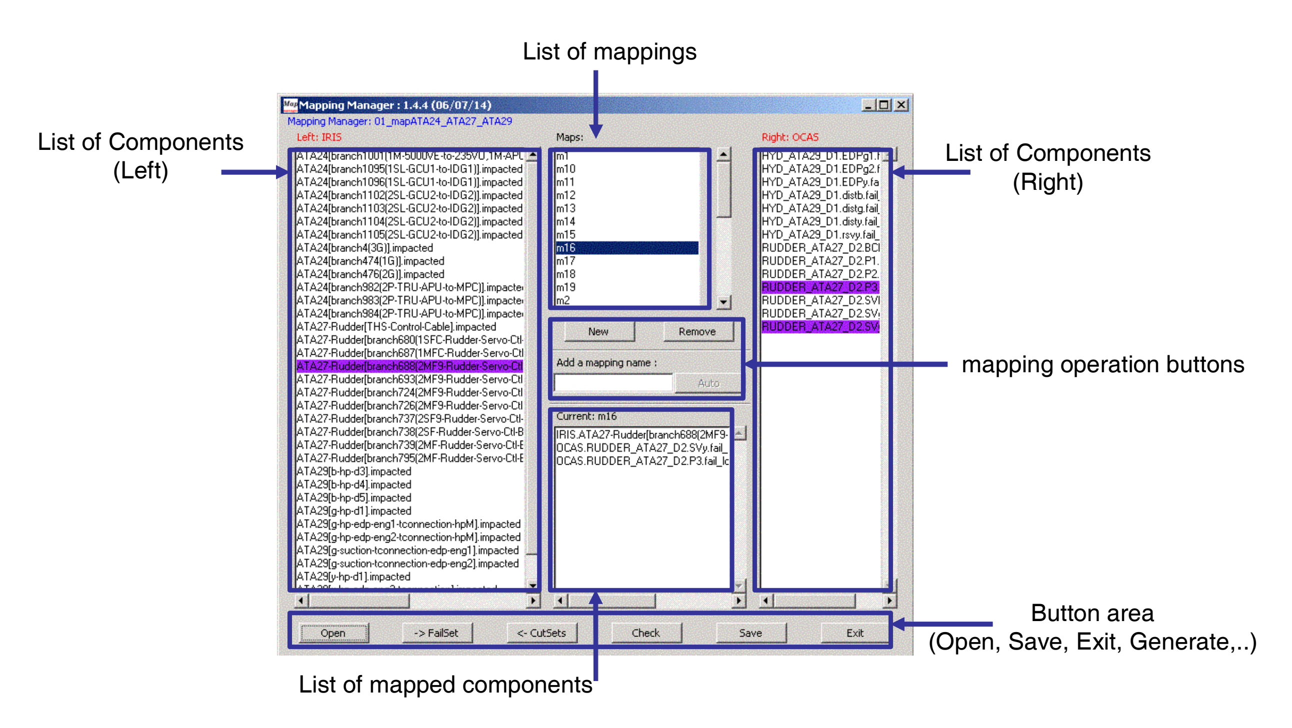
```
<cutsets failsets="Safety Results of the Hydraulic Functional Model
extended with tyre burst information" failmodes="Functional">
…
 <cutset original_size="2" name="a_cut_set_name">
    <failset name="a_fail_set_name">
      <failmode world="Geometrical" component="Y-EDP" fm="impacted"
description="Yellow Engine Driven Pump" relevant="yes">
      <failmode world="Geometrical" component="B-HP-distribution-1"
fm="impacted" " description="Blue High Pressure Distribution line (first
part)" relevant="no">
      <failmode world="Geometrical" component="B-HP-distribution-2"
fm="impacted" " description="Blue High Pressure Distribution line (second
part)" relevant="no">
     <failmode world="Geometrical" component="B-HP-distribution-3"
fm="impacted" " description="Blue High Pressure Distribution line (third
part)" >
  </failset>
</cutset>
…
</cutsets>
```

A textual format is also available for Geometrical cut set files. The file has 5 columns separated by commas:

```
Cutset_Name, failset?, world, component, failure mode
```

2 MapManager Operations

2.1 MapManager GUI overview

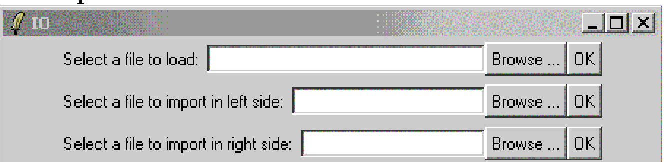


Main areas of MapManager GUI

The mapping manager graphical user is divided in three main areas: on the left side we find the list of components imported from the left-side world, on the right side is the list of components imported from the right-side world and in the middle area is shown the mapping between left and right components. The middle area is divided in three parts: at the top is the list of mapping names, then there is an area containing mapping operations (create, name, remove) and at the bottom is the list of mapped components that are currently being viewed. At the bottom of the GUI there are several buttons that can be used to load, save, import or check files, to generate failure sets and geometrical cut sets and to exit the application.

2.2 Import component list

To import a component list, you first have to click on the Open button. A dialog window will appear, select the file including the components and click on the OK button related with the side where you want to import the list of components.



MapManager Input/Output Window

You will also be requested to enter the name of the current mapping file.

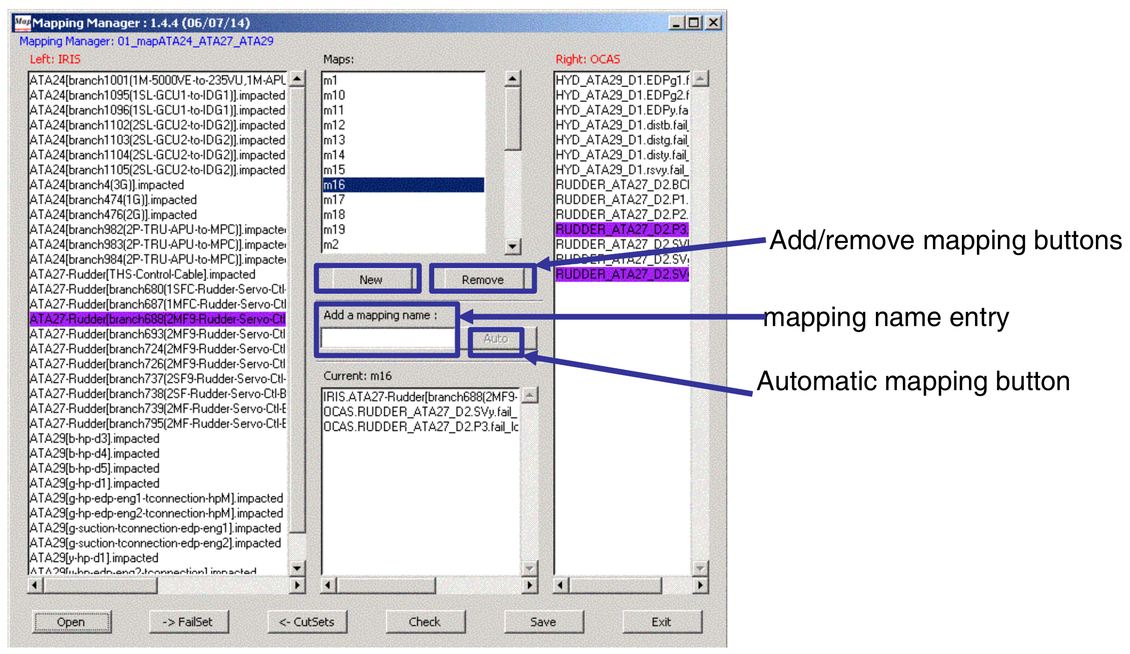


MapManager Mapping File Name window

The list of imported components should appear in the GUI.

2.3 Create manually a mapping

Once component lists are imported the definition of the mapping can start.



Buttons to create, name and remove a mapping

2.3.1. Create a mapping

To create a new mapping, click on the New button then a new mapping called map_untitled_$. This new mapping should appear in the mapping list.

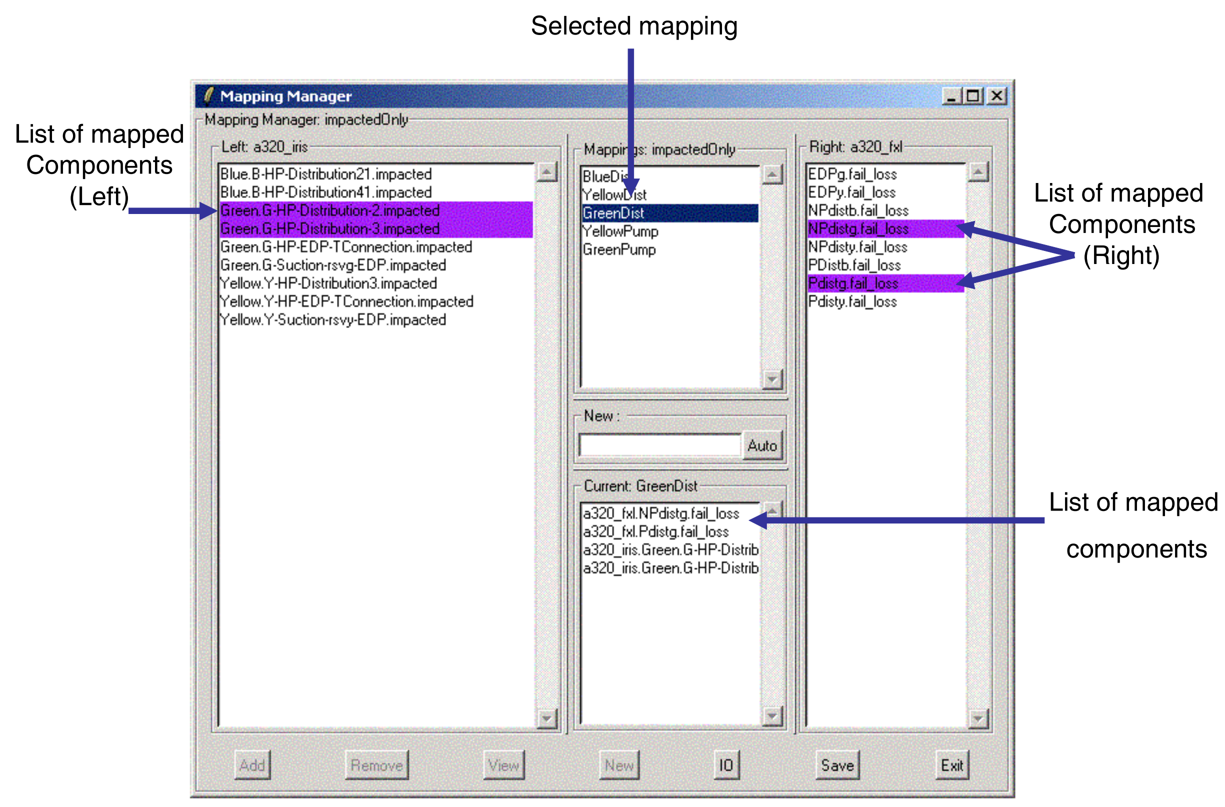
To create a new mapping with a name selected by the user, enter a mapping name in the mapping name entry and type on the RETURN key.The new mapping name should appear in the mapping list.

2.3.2. Add components to a mapping

Select a mapping by double-clicking on its name in the mapping list. Then add components from either side by double-clicking on them. Their names should appear on the list of mapped items and their names should be coloured in Purple in the component lists. It is also possible to select several components from one side and add all of them at a time by right-clicking.

2.3.3. Remove components from a mapping

Select a mapping by double-clicking on its name in the mapping list. Then remove components by double-clicking on their names in the list of mapped items. Their names should disappear from the list of mapped items and their names should no longer be coloured in Purple in the component lists.



Viewing a mapping

2.4 View a mapping

Select a mapping by double-clicking on its name in the mapping list. The names of mapped components should appear on the list of mapped items and their names should be coloured in Purple in the component lists.

2.5 Remove a mapping

Select a mapping by double-clicking on its name in the mapping list. Click on the Remove button to remove the mapping.

2.6 Save mappings into a file

Click on the Save button to save mappings into a file, a window will appear to help you select a file.

2.7 Load mappings from a file

Click on the Load button to load mappings from a file, a window will appear to help you select a file.

2.8 Check a set of mappings



Check mappings window

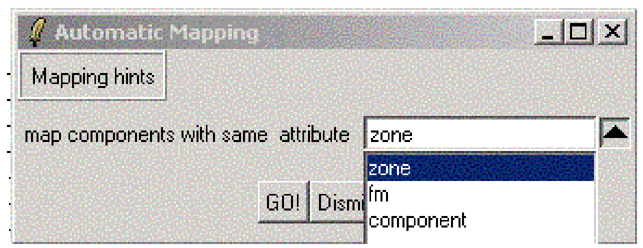
Click on the Check button to perform several sanity checks on the mappings that were created. The mapping manager detects unmapped items: components that do not belong to any mapping, empty mapped items: components that belong to a mapping that contains no other component, and multiple mapped items: components that belong to several mappings. If at least one of these situations is detected a pop-up window is created that provides the list of unmapped, empty-mapped and multiply mapped items with an indication of the mappings that might have to be corrected. A file called *mapping_name.log* is automatically created that contains all these data when the analysed mapping is called *mapping_name.xml*. The sanity check is also automatically performed whenever a mapping is saved.

2.9 Create Automatically mapping suggestions

Mappings can be proposed automatically when sufficient information is provided in the component lists. For instance, we could consider that the component description includes an extra attribute `zone` that indicates in which zone of the aircraft the component is located. This attribute can be used to group component failure modes together.

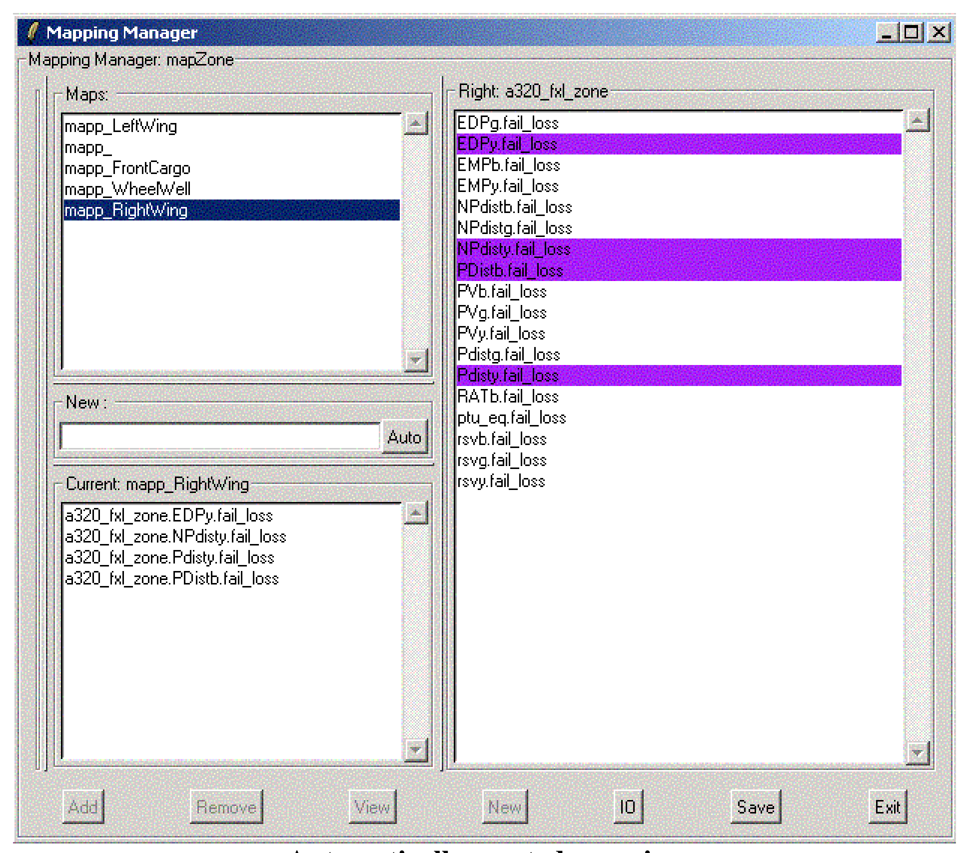
```
<failmodes name="a320_fxl_zone">
…
<failmode  component="PVg" fm="fail_loss" zone="WheelWell"/>
<failmode  component="EDPg" fm="fail_loss" zone="LeftWing"/>
<failmode  component="rsvg" fm="fail_loss" zone="WheelWell"/>
<failmode  component="NPdistg" fm="fail_loss" zone="LeftWing"/>
<failmode  component="Pdistg" fm="fail_loss" zone="LeftWing"/>
<failmode  component="EDPy" fm="fail_loss" zone="RightWing"/>
…
</failmodes>
```

To automatically create mapping suggestions click on the automatic mapping button, a window should appear that asks for the attribute to use to group components. Select an attribute in the list and click on the “Go!” button.



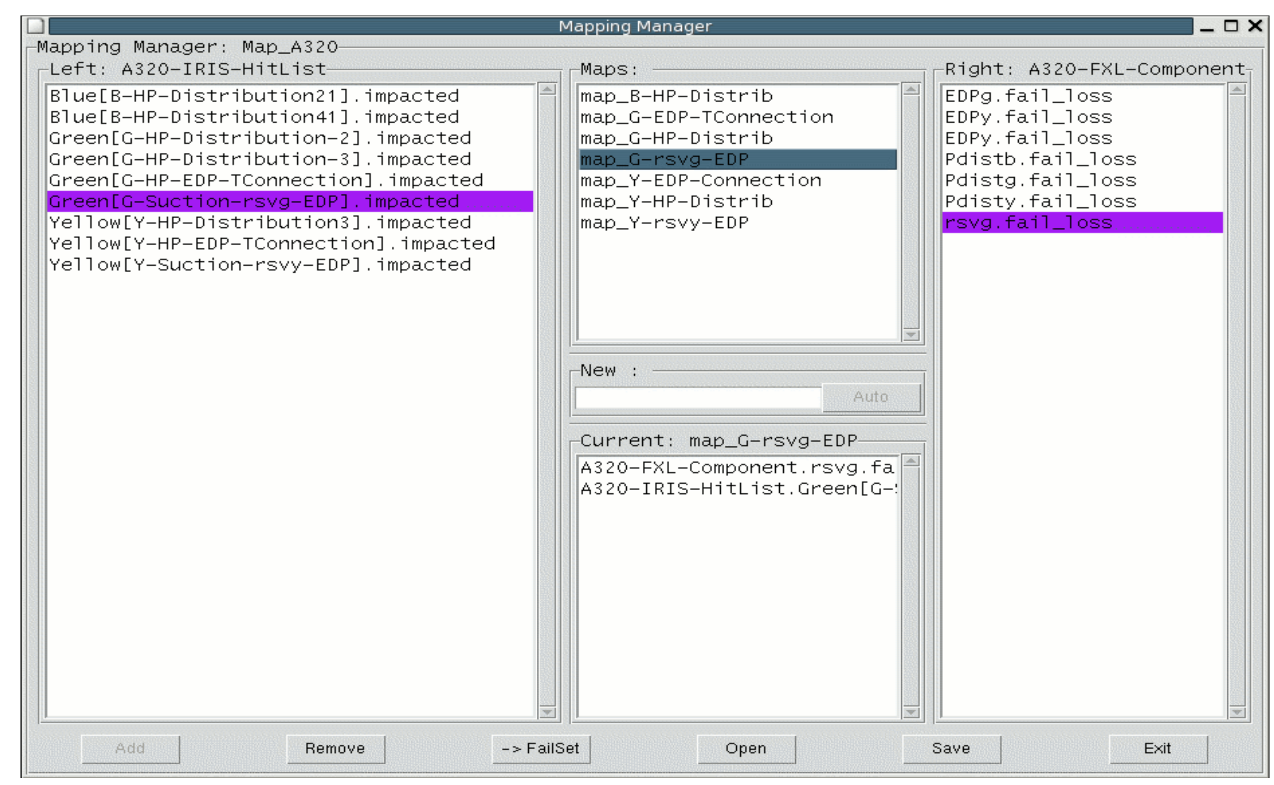
Selection of an attribute for Automatic mapping suggestion

A list of created mappings appears in the mapping list area, for each value V of the attribute a mapping `map_V` is created.



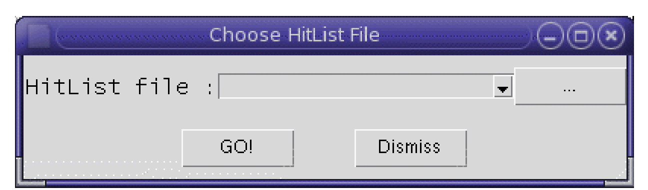
Automatically created mapping

2.10 Generate Fail Sets



-> Failset button

Once a mapping was defined or loaded, click on the button labelled "-> FailSet", a dialog window will appear that will let you either select the files loaded in the Right or Left side as a hit-list or select a hit-list stored elsewhere.



Failset Hitlist selection dialog

Click on the button labeled "GO!" when the hit list is selected. Then, another dialog window will appear to select the file name for the failsets. Once this file is selected, the fail sets file is generated.

Notice that for the failset generation to work properly the value of the `name` attribute of the hitlist should match the value of the `left` or `right` attribute of the currently loaded mapping file.

2.11 Generate Geometrical Cut Sets

Once a mapping was defined or loaded, click on the button labeled "<-CutSets"", a dialog window will appear that will let you select a safety assessment results file. Click on the button labeled "GO!" when the file is selected. Then, another dialog window will appear to select the file name for the geometrical cut sets. Once this file is selected, two geometrical cut sets files are generated: one using the XML format described in section 1.2 and the other file contains the same geometrical cut sets in a text format. If "a_geo_name.xml" is the name of the geometric cut set file selected by the user then "a_geo_name.cat.txt" is the name of the geometric cut sets in text format.

2.12 Mapping Manager commands

The mapping manager can be called using the following commands:

```
mapmanager [option file_name] [option parameter]

option -mp : loads file_name as a mappings file
option -hll : imports file_name as the Left component list
option -hlr : imports file_name as the Right component list
option -traj : if parameter=1 includes trajectory details in the generated
failset else (parameter=0) do not include trajectories details in the
failset
option -fseti : file_name is the hit list file used to generate failsets
option -fseto : file_name is the failsets file
option -cseti : file_name is the cut set file used to generate geo cut sets
option -cseto : file_name is the geo cut sets file
```

Bibliographie

[47696] ARP 4761. *Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipement.* Aerospace Recommended Practice, SAE, 1996.

[ABC94] A. Arnold, D. Bégay, et P. Crubillé. *Construction and analysis of transition systems with MEC.* World Scientific, 1994.

[adl00] armée de l'air. *Mirage 2000d : Suivi de terrain.* Rapport Technique, Armée de l'air, 2000.

[Air96] Airbus. *ABD200, Requirements and Guidelines for the systems designers*, 1996.

[Air00] Airbus. *IRIS, Basic 3D Design User's Manual*, 2000.

[APGR00] A. Arnold, G. Point, A. Griffault, et A. Rauzy. *The altarica formalism for describing concurrent systems. Fundamenta Informaticae*, 40 :109–124, 2000.

[AW05] AADL-WGL. *AADL Error Model Annex.* 2005.

[BAK04] C. Betous-Almeida et K. Kanoun. *Construction and stepwise refinement of dependability models. Performance Evaluation*, 56(1-4) :277–306, 2004.

[BBC+08] P. Bieber, J.P Bodeveix, C. Castel, D. Doose, M.Filali, F. Minot, et C. Pralet. *Constraint-based design of avionics platform - preliminary design exploration.* Dans *ERTS08*, 2008.

[BCMD90] J. R. Burch, E. M. Clarke, K. L. McMillan, et D. L. Dill. *Sequential circuit verification using symbolic model checking.* Dans *DAC*, pages 46–51, 1990.

[BLED99] N. Blackwell, S. Leinster-Evans, et S. Dawkins. *Developing safety cases for integrated flight systems.* Dans *IEEE Aerospace Conference.* IEEE, 1999.

[BMT99] A. Bondavalli, I. Mura, et K.S. Trivedi. *Dependability modelling and sensitivity analysis of scheduled maintenance systems. 3rd European Dependable Computing Conference (EDCC-3)*, pages 7–23, 1999.

[BR94] S. Brlek et A. Rauzy. *Synchronization of constrained transition systems.* Dans *Proceedings of the First International Symposium on Parallel Symbolic Computation*, pages 54–62. World Scientific Publishing, 1994.

[Bry86] R. E. Bryant. *Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers*, 35(8) :677–691, 1986.

[BV06] B. Berthomieu et F. Vernadat. *Time petri nets analysis with tina.* Dans *QEST '06 : Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems*, pages 123–124, Washington, DC, USA, 2006. IEEE Computer Society.

[BVÅ+03] M. Bozzano, A. Villafiorita, O. Åkerlund, P. Bieber, C. Bougnol, E. Böde, M. Bretschneider, A. Cavallo, C. Castel, M. Cifaldi, A. Cimatti, A. Griffault, C. Kehren, B. Lawrence, A. Lüdtke, S. Metge, C. Papadopoulos, R. Passarello, T. Peikenkamp, P. Persson, C. Seguin, L. Trotta, L. Valacca, et G. Zacco. *Esacs : an integrated methodology for design and safety analysis of complex systems. ESREL*, 2003.

[CCGR00] A. Cimatti, E. Clarke, F. Giunchiglia, et M. Roveri. *Nusmv : a new symbolic model checker*, 2000.

[CHD+04] H. Cambazard, P-E. Hladik, A-M. Déplanche, N. Jussien, et Yvon Trinquet. *Décomposition et apprentissage pour un problème d'allocation de tâches temps-réel.* Dans *10e Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'04)*, pages 123–138, Angers, France, 2004.

[Cle07] ClearSy. *Manuel de référence du langage B.* Support Atelier B, Février 2007. `http://www.atelierb.societe.com/documents.htm`.

[Coo71] S. A. Cook. *The complexity of theorem-proving procedures.* Dans *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[CPR04] F. Cassez, C. Pagetti, et O. Roux. *A timed extension for altarica. Fundam. Inf.*, 62(3-4) :291–332, 2004.

[DAS85] DASSAULT Systems. *CATIA, Basic 3D Design User's Manual*, 1985.

[DPS+08] X. Dumas, C. Pagetti, L. Sagaspe, P. Bieber, et P. Dhaussy. *Vers la génération de modèles de sûreté de fonctionnement.* Dans *conférences LMO'8 et CAL'08*, Mars 2008.

[EC97] A. Evans et T. Clark. *Foundations of the Unified Modeling Language.* Dans *Proc. of the 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, 23-24 September 1997*, 1997.

[EH82] E. A. Emerson et J. Y. Halpern. *Decision procedures and expressiveness in the temporal logic of branching time.* Dans *In Proceedings of the Fourteenth Annual ACM Symposium on theory of Computing, STOC '82*, San Francisco, California, United States, May 05 - 07 1982. ACM. New York, NY, 169-180.

[EL86] E. A Emerson et C. Lei. *Model checking in the propositional mu-calculus.* Rapport Technique, University of Texas at Austin, Austin, TX, USA, 1986.

[ES03] N. Eén et N. Sörensson. *An extensible sat-solver.* Dans *SAT*, pages 502–518, 2003.

[FGH06] P. H. Feiler, D. P. Gluch, et J. J. Hudak. *The Architecture Analysis & Design Language (AADL) : An Introduction.* Rapport Technique, Software Engineering Institute (SEI), http ://www.sei.cmu.edu/publications/documents/06.reports/06tn011.html, 2006.

[FLV03] P. H. Feiler, B. Lewis, et S. Vestal. *The SAE architecture analysis & design language (AADL) standard : A basis for model-based architecture driven embedded systems engineering.* Dans *RTAS 2003 Worshop on Model-Driven Embedded Systems*, Washington, D.C., May 2003. IEEE.

[FP93] L. Fribourg et M. Veloso Peixoto. *Concurrent constraint automata.* Dans *ILPS '93 : Proceedings of the 1993 international symposium on Logic programming*, page 656, Cambridge, MA, USA, 1993. MIT Press.

[Gob02] R. Gobard. *Traduction altarica vers lustre.* Rapport Technique, LaBRI, 2002.

[GT04] E. Giunchiglia et A. Tacchella, éditeurs. *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 de *Lecture Notes in Computer Science*. Springer, 2004.

[HCDJ06] P-E. Hladik, H. Cambazard, A-M. Déplanche, et N. Jussien. *Guiding architectural design process of hard real-time systems with constraint programming.* Dans *Third Taiwanese-French Conference on Information Technology (TFIT 2006)*, pages 317–331, Nancy, France, 2006.

[HCDJ08] P-E. Hladik, H. Cambazard, A-M Déplanche, et N. Jussien. *Solving a real-time allocation problem with constraint programming. J. Syst. Softw.*, 81(1) :132–149, 2008.

[KB96] K. Kanoun et M. Borrel. *Dependability of Fault-Tolerant Systems-Explicit Modeling of the Interactions between Hardware and Software Components. Proc. IEEE International Computer Performance & Dependability Symp.(IPDS'96)*, 49 :252–61, 1996.

[KSB+04] C. Kehren, C. Seguin, P. Bieber, C. Castel, C. Bougnol, J.-P. Heckmann, et S. Metge. *Advanced simulation capabilities for multi-systems with altarica.* Dans *International System Safety Conference*, 2004.

[KSBC04] C. Kehren, C. Seguin, P. Bieber, et C. Castel. *Analyse des exigences de sûreté d'un système électrique par model-checking.* Dans *Actes du Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement*, pages 492–497, Bourges, France, Octobre 2004.

[LaB07] LaBRI. *ARC : AltaRica Checker.* http ://altarica.labri.fr/, 2007.

[Lap96] J-C. Laprie. *Guide de la sûreté de fonctionnement*, 1996. Cépadues, ISBN : 2854283821.

[Lew06] B. Lewis. *The sae architecture analysis & design language (aadl) a standard for engineering performance critical systems. ERTS'06*, January 2006.

[LM88] C. Lin et D. C. Marinescu. *Stochastic high-level petri nets and applications. IEEE Trans. Comput.*, 37(7) :815–825, 1988.

[McM98] K. L. McMillan. *The SMV language*, Mars 1998. disponible à : `http ://www-cad.eecs.berkeley.edu/~kenmcmil/psdoc.html`.

[MLMEP+05] Jr. M. L. McKelvin, G. Eirea, C. Pinello, S. Kanajan, et A. L. Sangiovanni-Vincentelli. *A formal approach to fault tree synthesis for the analysis of distributed fault tolerant systems.* Dans *EMSOFT '05 : Proceedings of the 5th ACM international conference on Embedded software*, pages 237–246, New York, NY, USA, 2005. ACM.

[Pag04] C. Pagetti. *Extension temps réel d'AltaRica.* PhD thesis, Ecole Centrale de Nantes, April 2004.

[PBR03] T. Petit, C. Bessière, et J-C. Régin. *Détection de conflits pour la résolution de problèmes sur-contraints.* Dans *9emes Journées nationales sur la résolution pratique de problèmes NP-complets (JNPC'03)*, pages 293–307, Amiens, France, 2003.

[Pet81] J. Lyle Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[PL00] P. Pettersson et K. G. Larsen. *UPPAAL. Bulletin of the European Association for Theoretical Computer Science*, 70 :40–44, 2000. `http://www.uppaal.com/`.

[Pnu77] A. Pnueli. *The temporal logic of programs.* Dans *18th IEEE Symp*, pages 46–57. Foundations of Computer Science, 1977.

[Poi00] G. Point. *AltaRica : contribution à l'unification des méthodes formelles et de la sûreté de fonctionnement.* PhD thesis, LaBRI, Université Bordeaux 1, January 2000.

[PR99] G. Point et A. Rauzy. *Altarica - constraint automata as a description language. European Journal on Automation*, 1999. Special issue on the *Modelling of Reactive Systems.*

[Rau] A. Rauzy. Altarica ToolBox : Combava*. http ://www.arboost.com/altarica-page.htm.*

[Rau02] A. Rauzy. Mode automata and their compilation into into fault trees. Reliability Engineering and System Safety*, 78 :1–12, 2002.*

[RKK06] A-E. Rugina, K. Kanoun, et M. Kaâniche. Modélisation de la sûreté de fonctionnement de système à partir du langage aadl. Rapport LAAS*, 2006.*

[Rug05] A. E. Rugina. System dependability evaluation using aadl. *Dans* 1iere Rencontres Jeunes Chercheurs en Informatique Temps Réel (RJCITR'2005)*, September 2005.*

[RY97] A. Rauzy et Y.Dutuit. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. Reliability Engineering and System Safety*, 58, 1997.*

[SA06] SAE-AS5506/1. Architecture analysis and design language annex volume 1. Error Model Annex*, 2006.*

[SB07] L. Sagaspe et P. Bieber. Constraint-based design and allocation of shared avionics resources. *Dans* 26th Digital Avionics Systems Conference*, 2007.*

[Tho06] P. Thomas. Générateur de séquences et autres outils basés sur un stepper*. Rapport Technique, Dassault Aviation, 2006.*

[Ver06] T. Vergnaud. Modélisation de systèmes temps-réels répartis embarqués pour la génération automatique d'applications formellement vérifiées*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, December 2006.*

[Vin03] A. Vincent. Conception et réalisation d'un vérificateur de modèles AltaRica*. PhD thesis, LaBRI, Université Bordeaux 1, décembre 2003.*

[Yam95] S. Yamane. Verification system for real-time specification based on extended real-time logic. *Dans* RTCSA '95 : Proceedings of the 2nd International Workshop on Real-Time Computing Systems and Applications*, page 192, Washington, DC, USA, 1995. IEEE Computer Society.*